Títol: Bases de dades basades en columnes

Volum: 1
Alumne: Emili Calonge Sotomayor

Director/Ponent: David Carrera Pérez
Departament: DAC
Data: 20/01/2010

DADES DEL PROJECTE

Títol del Projecte:	Bases de dades basades en columnes
Nom de l'estudiant:	Emili Calonge Sotomayor
Titulació:	Enginyeria en Informàtica
Crèdits:	37.5
Director/Ponent:	David Carrera Pérez
Departament:	DAC

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	Yolanda Becerra Fontal
Vocal:	Elvira Guardia Manuel
Secretari:	David Carrera Pérez

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data: 20 de gener de 2011

Bases de dades basades en columnes

Emili Calonge Sotomayor

20 de gener de 2011

Índex

Agraïments

Vull donar les gràcies a la gent que m'ha ajudat a portar aquest projecte endavant, en especial la meva família i el meu soci. També m'agradaria mencionar al director del projecte, que m'ha donat l'oportunitat de descobrir temes nous i de desenvolupar aquest projecte amb total llibertat.

Per últim m'agradaria fer una menció especial a la gent de Cloudera, que m'han donat accés a material privat dels seus cursos, i s'han interessat en el desenvolupament d'aquest projecte.

Capítol 1

Introducció

1.1 Motivació

Al anar arribant al final de la carrera, moltes vegades em plantejava què fer com a pfc. Una idea que sempre em venia al cap era la d'aprofitar el pfc per a realitzar alguna aplicació per a la feina i documentar-ne el procés. Però realment em sembla un tipus de projecte molt poc ambiciós i molt avorrit. No es que tingui res en contra de desenvolupar aplicacions, de fet és el que m'agrada, és al que em dedico i espero poder-m'hi dedicar durant la meva vida professional. És simplement que desenvolupar aplicacions ja ho se fer, no em representa cap repte i per tant no em motiva. I dedicar les 700 hores que s'han de dedicar a un pfc fent una cosa que no em motiva, em sembla una pèrdua de temps. Per tant fa molt de temps que he provat de buscar què volia fer com a pfc.

Gairebé totes les idees que tenia es centraven en un tema, les aplicacions web d'alta demanda, és a dir, aplicacions web que son usades per milions d'usuaris, amb tot el que això comporta, la infraestructura de servidors necessària, la replicació, l'alta disponibilitat, etc.

Des de ja fa un temps que cada dia és més freqüent sentir a parlar d'aplicacions que tenen milions d'usuaris, que generen PetaBytes d'informació, que necessiten "granges"de servidors per a poder funcionar i garantir la seva disponibilitat. I més enllà de l'èxit que suposa per als seus creadors, el que sempre em volta pel cap quan en sento parlar, és una única pregunta ¿Com ho fan?

Per moltes assignatures que hagi fet, per molt que hagi treballat un temps administrant sistemes, i per molt que hagi llegit sobre el tema, desconec com fan aquestes empreses per a fer funcionar les seves aplicacions. Com es pot garantir que entri la quantitat de gent que entri, l'aplicació sempre funcioni?

Com ho fan per si cauen una sèrie de servidors, no perdre la informació? Com fan per mantenir temps de resposta ràpids sigui quina sigui la càrrega del sistema? Son moltes de les preguntes que em venen al cap quan penso en aquestes aplicacions.

Així que quan a l'últim quadrimestre vaig cursar CARS i a l'assignatura es tractaven temes relacionats amb l'escalabilitat, la distribució, etc. em vaig acabar de decidir per enfocar el treball cap aquest tema. El primer aspecte que em feia falta resoldre era trobar el professor que em portés el projecte, ho vaig proposar als professors de l'assignatura i em van dir que si.

Una vegada triat el professor, encara em faltava triar el tema. A la primera reunió, es van proposar una llista d'idees. Una d'elles va ser desenvolupar un sistema que fos capaç de generar els recursos necessaris per a suportar una aplicació d'alta demanda. L'objectiu era crear un *clúster* de servidors que fos autosuficient, que fos capaç d'auto escalar-se segons la demanda. És a dir, en hores amb poca demanda tenir un o dos servidors, però a la que creixés la demanda, ràpidament afegir servidors al clúster per a satisfer la demanda. L'objectiu era poder garantir temps de resposta acceptables sense necessitat d'intervenció humana.

Així que vaig començar amb aquest projecte, però era difícil de portar a terme, perquè feia falta infraestructura per a poder realitzar les proves, i calia dedicar molt de temps a administrar sistemes, i no era una cosa que em generés massa motivació. Així que poc a poc va anar decaient la il·lusió, fins que un dia parlant, el director del projecte em va parlar de les bases de dades orientades a columnes. Jo no n'havia sentit a parlar mai, no tenia ni idea de què eren o per a que servien, però pel poc que em va explicar, em va semblar un tema interessant. Així que després d'investigar molt per sobre el tema, vam veure que donava prou de si com per a fer-ne un pfc. I sobretot, jo vaig veure que aquest era un pfc que realment em motivava i que em venia de gust fer.

Amb la primera investigació que vam fer abans de començar el projecte el que vam trobar van ser sobretot les motivacions de les bases de dades orientades a columnes. I aquesta era bàsicament oferir sistemes d'emmagatzemament per a entorns d'alta demanda. Bases de dades orientades únicament a aquests entorns i de les quals els seus punts forts eren:

- Escalabilitat
- Replicació
- Alta disponibilitat

Eren uns sistemes dissenyats per a cobrir les necessitats exactes dels sistemes

d'alta demanda, donat que els sistemes actuals tenien certes limitacions que feia complicat mantenir aquestes aplicacions.

Així que amb aquest primer cop de vista, aquest tema complia tots els meus requisits. Un tema del que no sabia res i per tant em suposava un repte. Sobre un tema que era exactament el tema que havia volgut des d'un principi, les aplicacions d'alta demanda. I sobre un tema en el que sempre he tingut bastant d'interès, les bases de dades. A part de complir totes les meves necessitats, també era un tema en el que el grup de recerca del director del projecte estava interessat, per tant tothom sortia guanyant. I amb aquesta poca informació sobre la taula vam decidir començar el projecte que ara us presento.

1.2 Objectius

Com que al començar el projecte no tenia un coneixement gaire ampli sobre el tema, em va ser difícil establir una llista d'objectius concrets des d'un inici. Així que la llista d'objectius ha anat creixent a mida que ha anat avançant el projecte i he anat adquirint més coneixement sobre el tema. Com anirem veient al llarg del projecte alguns dels objectius inicials s'han descartat per haver donat un enfoc inicialment que no era el més adient, altres han aparegut al veure temes interessants que podia explorar i d'altres simplement no he tingut temps per a realitzar-los.

Així que en aquest apartat mostro la llista complerta dels objectius que hem anat proposant des d'el inici de projecte i a les conclusions dels diferents apartats aniré posant en situació cadascun d'ells.

Per començar, es van definir els objectius principals a alt nivell:

1. Contextualització de les bases de dades orientades a columnes respecte els sistemes relacionals

2. Estudi dels sistemes de bases de dades orientades a columnes

3. Desenvolupament d'una aplicació per veure el funcionament de les bases de dades orientades a columnes.

Una vegada definits aquests objectius globals, es van subdividir en petits objectius més concrets.

Així per exemple, del primer objectiu en van sorgir:

1. Contextualització de les bases de dades orientades a columnes respecte els sistemes relacionals

   (a) Resum de les característiques dels sistemes relacionals
   (b) Explicació de les limitacions dels sistemes relacionals
   (c) Descripció dels sistemes orientats a columnes

El segon objectiu, una vegada acabat el primer es va concretar més, i l'estudi dels sistemes orientats a columnes el vam centrar en estudiar dos dels molts productes que existeixen actualment. Així els objectius van passar a ser:

2. Estudi dels sistemes de bases de dades orientades a columnes
   (a) Estudi de les característiques dels sistemes orientats a columnes
   (b) Estudi a fons d'HBase
   (c) Estudi a fons de Cassandra

El tercer objectiu és el que ha sofert més canvis al llarg del projecte, al iniciar el projecte la llista era:

3. Desenvolupament d'una aplicació per veure el funcionament de les bases de dades orientades a columnes.
   (a) Desenvolupament d'una aplicació que faci ús d'HBase
   (b) Desenvolupament d'una aplicació que faci ús de Cassandra

Una vegada el projecte va anar avançant es va veure que desenvolupar una aplicació no era el més adient ja que suposava massa esforç i el que realment ens interessava era centrar-nos únicament en la capa de la base de dades, així els objectius van evolucionar cap a:

3. Desenvolupament d'una aplicació per veure el funcionament de les bases de dades orientades a columnes.
   (a) Estudi del funcionament de Twissanra, una aplicació que funciona sobre Cassandra
   (b) Adaptació de Twissandra per a usar HBase
   (c) Instrumentació de Twissandra per a desar logs a HBase
   (d) Ús d'eines d'anàlisi de dades sobre HBase com Hive o Pig

El tercer i quart objectius van aparèixer mentre portava a terme el segon i vaig decidir que no era l'enfoc adequat.

Per últim, amb tota la part inicial de documentació acabada i veient el temps que quedava, es va decidir ser una mica més ambiciós i provar de fer un segon cas pràctic més extens, per al qual es van proposar els següents objectius:

3. Desenvolupament d'una aplicació per veure el funcionament de les bases de dades orientades a columnes.

   (a) Desenvolupar un generador de càrrega basat en Twitter sobre Twissandra

   (b) Obtenir dades de Twitter mitjançant les API's

   (c) Extreure models de comportament sobre les dades obtingudes de Twitter

   (d) Generar càrrega seguint els models obtinguts

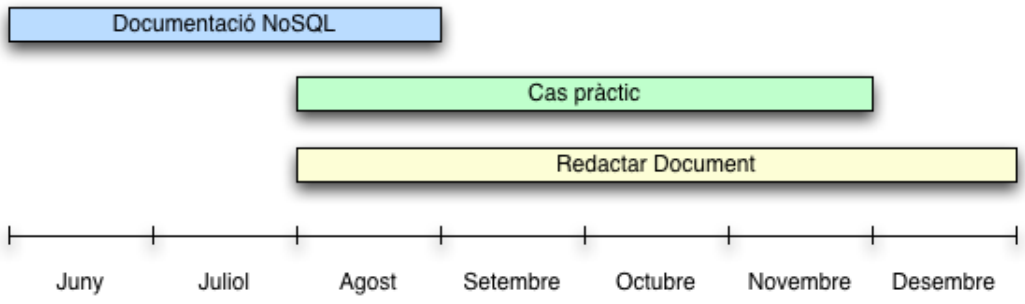
   (e) Analitzar el rendiment de Cassandra, fent proves de càrrega sobre Twissandra

   (f) Modificar el model de dades de Twissandra per a veure les diferències en el rendiment global de l'aplicació

   (g) Analitzar els logs de Twissandra deixats sobre HBase

1.3 Planificació

Aquest projecte el vaig començar al Maig del 2010. Al principi buscàvem a veure de què podia fer el projecte i com he explicat abans, primer vaig començar a realitzar un projecte sobre sistemes autoescalables. Amb aquest primer projecte hi vaig estar durant el Maig, però a finals de Maig ja vaig decidir canviar de tema i començar aquest sobre bases de dades orientades a columnes. Com que al principi desconeixia totalment el tema no vaig poder marcar una planificació, sinó que van passar algunes setmanes fins que vaig començar a veure clar quins eren els objectius que em volia marcar i quina seria la planificació del projecte.



Listing 1: Planificació inicial del projecte

La figura 1 mostra la planificació que vaig marcar un cop m'havia documentat una mica, cap a la tercera setmana de Juny. Com es pot veure, vaig decidir primer documentar-me àmpliament, i després realitzar una aplicació que fes ús de les bases de dades NoSQL. Com hem vist als objectius del projecte, aquesta planificació ha sofert molts canvis durant el temps que he estat fent el projecte. Al final del projecte mostraré les diferències entre aquesta primera planificació, i les etapes en les que he realitzat realment el projecte.

1.4 Background Tècnic

L'objectiu d'aquest projecte és entendre què son les bases de dades basades orientades a columnes, per a què serveixen i per a què no, quins son els seus punts forts i quins els seus punts febles. Per tant el primer que hem de fer és posar en context aquests sistemes amb el sistema més usat actualment, les bases de dades relacionals.

Amb la globalització d'Internet, poc a poc han anat apareixent el que anomenem aplicacions d'alta demanda. Aplicacions que son usades concurrentment per milers (o milions) d'usuaris. Per a poder mantenir aquestes aplicacions, el primer que es va haver de fer va ser oferir la capacitat de càlcul suficient i l'ample de banda suficient per abastir la demanda. Però una vegada complerts aquests requisits, la següent necessitat va ser emmagatzemar tota la informació generada. Un dels bens més preuats per qualsevol empresa és la informació, no només la informació estrictament necessària per a fer funcionar els seus sistemes, sinó qualsevol tipus d'informació derivada. Logs dels servidors, patrons d'ús dels usuaris, etc. El tractament d'aquesta informació és el que permet extreure conclusions i prendre decisions per a evolucionar els sistemes, es detecten necessitats i es supleixen.

Així que durant bastant de temps les empreses es van adaptar a les solucions existents, bàsicament els sistemes de bases de dades relacionals. Però degut a les limitacions d'aquestes abans exposades, amb el temps van acabar sorgint uns sistemes dissenyats explícitament per a suplir les necessitats de les aplicacions d'alta demanda.

Els principals objectius a l'hora de dissenyar els nous sistemes van ser:

- Replicació
- Escalabilitat
- Model de dades flexible

Es buscava un sistema en que afegir nodes al sistema de bases de dades fos una tasca trivial i augmentés linealment les capacitats del sistema. On la

replicació fos part del sistema, que funcionés sense necessitat d'intervenció humana i que permetés garantir la disponibilitat de les dades davant de qualsevol imprevist. I per últim, oferir un model de dades flexible, on la majoria de canvis es poguessin fer instantàniament i el seu cost fos el mínim possible.

Per altra banda, amb el model de dades el que es buscava era oferir un model realment flexible, evitant el model estàtic relacional. Un dels grans avantatges del model relacional és a l'hora una de les seves grans limitacions, els índexs. Un índex comporta complexitat i cost a l'hora de modificar els models. Per tant el model orientat a columnes fa un canvi radical en aquest aspecte i ofereix un model clau:valor. On un valor s'identifica per una única clau. Per a obtenir un valor ho farem sempre usant una única clau, aquest comportament és anàleg a una taula de Hash.

Cadascuna de les diferents implementacions dels sistemes orientats a columnes aborden aquests punts de maneres diferents, però aquests son els punts en comú que els uneixen i els identifiquen.

Aquestes son a grans trets les característiques principals dels sistemes orientats a columnes, però les diferències amb els sistemes relacionals van molt més enllà d'aquestes característiques. Per exemple, els sistemes relacionals (la majoria) es poden consultar amb el llenguatge SQL i els orientats a columnes no, els sistemes relacionals (la majoria) garanteixen les propietats ACID i els orientats a columnes no, etc.

Al llarg del document les anirem mostrant totes i aprofundirem en les més rellevants de cara a aquest projecte.

Capítol 2

Sistemes relacionals i sistemes orientats a columnes

2.1 Sistemes de bases de dades Relacionals

Dins del món de la computació, no hi ha dubte que la manera més freqüent d'emmagatzemar informació, és usant bases de dades. Les bases de dades ens permeten desar grans quantitats d'informació, i obtenir aquesta informació de manera fàcil i flexible mitjançant llenguatges com el SQL. La gran majoria dels programes informàtics actuals fan servir bases de dades relacionals per a desar la informació.

El món de la computació no para d'avançar, per una banda cada vegada els processadors son més ràpids, i per altra banda, la quantitat d'informació que es pot emmagatzemar és més gran. Així per exemple fa uns anys, una empresa en podia tenir prou amb un disc dur d'uns quants MegaBytes, mentre que ara la unitat bàsica per a qualsevol entorn, gira en torn dels GigaBytes. I estem parlant d'empreses petites, o entorns familiars qualsevol. Amb aquesta evolució, les demandes d'emmagatzemament han canviat notòriament, i les bases de dades s'han anat adaptant a aquestes demandes dins de les seves possibilitats.

Però estem arribant a un moment en que algunes situacions comencen a forçar els límits suportats per les bases de dades relacionals. L'accés generalitzat a Internet, fa que una empresa pugui oferir els seus serveis a milions d'usuaris, gràcies a la globalització d'aquest medi. Això fa que si una aplicació té èxit, fàcilment pugui tenir milions d'usuaris. Ens podem trobar doncs, amb una aplicació on concurrentment tinguem connectats a milions d'usuaris, que constantment estan fent lectures i escriptures a la base de dades.

Aquests entorns eren impensables quan es van dissenyar els SGBDR's [1] , i això fa que adaptar el seu ús per a aquests entorns no sigui una tasca fàcil. I tot i que es pot aconseguir i moltes organitzacions encara fan servir SGBDR's en aquest tipus d'entorn, han començat a aparèixer eines especialitzades en aquests tipus d'entorns.

2.1.1 Repàs sobre els sistemes de bases de dades relacionals

Els sistemes de bases de dades relacionals, son uns sistemes àmpliament coneguts, ja que son l'estàndard actual en quant a bases de dades. El punt fort d'aquests sistemes és la manera en que ens permeten emmagatzemar les dades, ja que ens ofereixen maneres molt flexibles de recuperar-les posteriorment, i ens ofereixen garanties sobre aquestes dades.

Propietats ACID

Quan parlem de bases de dades, parlem d'emmagatzemar dades. I quan parlem de dades, normalment parlem de la informació més valuosa que una empresa posseeix. Per tant és essencial que una base de dades ens garanteixi una sèrie de propietats sobre les dades que emmagatzema. Aquestes propietats bàsiques tenen les sigles ACID [2].

Atomicitat Aquesta propietat ens garanteix que tot el contingut d'una transacció serà executat. En cas que només es pugui executar una part de la transacció per qualsevol raó, aquesta no s'executarà i no modificarà l'estat de la base de dades.

Consistència La consistència ens garanteix que que qualsevol transacció portarà la base de dades d'un estat consistent a un altre estat consistent. Si qualsevol part de la transacció provoqués una inconsistència, aquesta transacció no s'executaria. Un estat inconsistent pot fer referència per exemple a una equivocació al tipus de dades d'una columna.

Aïllament L'aïllament prevé que una transacció accedeixi a informació encara no confirmada per una altra transacció.

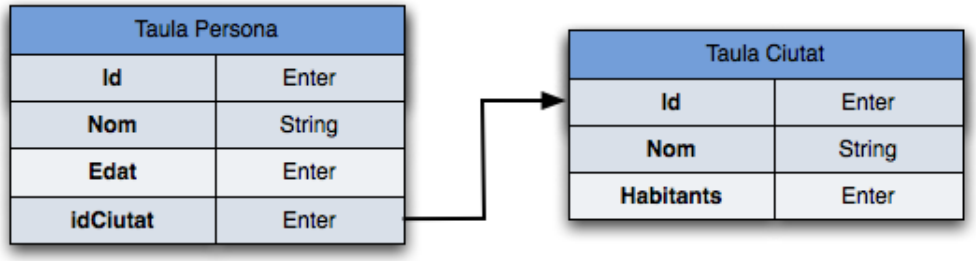
[1] Sistemes Gestors de Bases de Dades Relacionals
[2] Atomicity, Consistency, Isolation, Durability

Durabilitat La durabilitat ens garanteix que una vegada una transacció es confirma, les modificacions fetes per aquesta no es perdrà. Inclús en cas de caiguda del servidor, quan aquest es torni a aixecar, la informació encara hi serà.

Aquest conjunt de propietats son una de les claus de l'èxit dels sistemes relacionals, ja que donen garanties respecte a les dades que emmagatzemen. Sense aquestes garanties segurament moltes empreses no farien servir els SGBDR's.

Model de dades

Els sistemes relacionals ens permeten declarar taules formades per columnes, per a cada columna podem definir el tipus de dada que contindrà (numèric, text, etc.). Sobre aquestes columnes podem definir relacions, aquestes relacions ens permeten crear una estructura global de les dades.



Listing 2: Estructura de taules amb el model relacional

Com podem veure a la figura 2, podem definir dues taules on cada columna té un tipus de dada, i una columna d'una taula pot fer referència a una columna de l'altre. Amb aquest esquema podem fàcilment aconseguir un llistat de totes les persones i les seves poblacions de residència. El gran punt fort dels sistemes relacionals, és el llenguatge que ens ofereixen per a interactuar-hi. Aquest llenguatge s'anomena SQL [3] i és clarament l'estàndard en quant a llenguatges per a treballar amb bases de dades.

[3]Structured Query Language

```
CREATE  TABLE Persona ( id INT NOT NULL , nom VARCHAR(45) ,
    edat INT, idCiutat INT NULL , PRIMARY KEY (id) );
CREATE  TABLE Ciutat (id INT NOT NULL , nom VARCHAR(45),
    habitants INT, PRIMARY KEY (id) );
ALTER TABLE persona ADD CONSTRAINT fk_Persona_Ciutat
    FOREIGN KEY (idCiutat) REFERENCES ciutat(id)
```

Listing 3: Consultes per a la creació de taules en SQL

Amb les senzilles consultes que podem veure a la figura 3, crearíem les dues taules i la relació entre elles.

```
Select nom, edat From Persona Where edat > '20';
```

Listing 4: Exemple de consulta SQL

Nom	Edat
Joan	25
Maria	27

Listing 5: Resultat de la consulta

I amb una simple consulta com la de la figura 4 podriem obtenir un resultat com el de la figura 5.

Aquest tipus de model de dades és perfecte per emmagatzemar l'informació de qualsevol aplicació desenvolupada amb orientació a objectes. Ja que cada objecte correspon a una taula, i les relacions entre objectes es guarden amb les relacions del model relacional.

Índexs

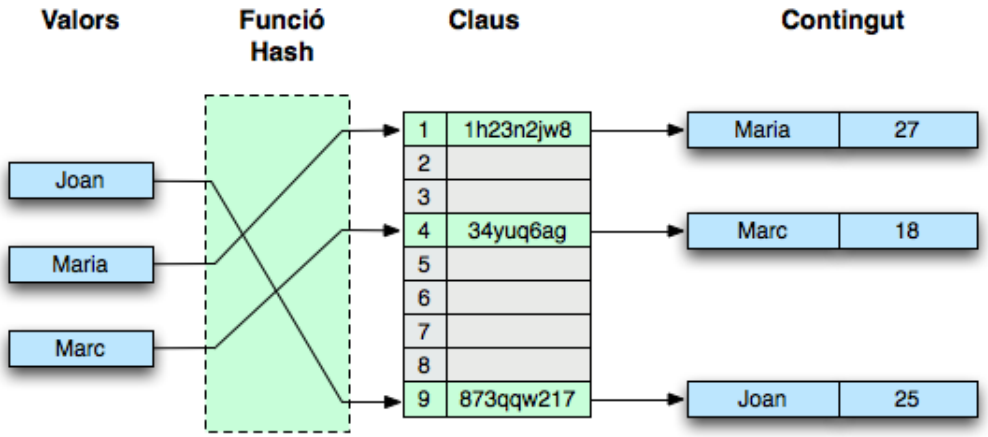
Una de les eines que fan més útils els SGBDR's és l'ús d'índexs. Els índexs el que ens permeten és optimitzar l'accés a les dades per a disminuir notablement els temps de cerca.

Si agafem per exemple la consulta de la figura 4, podem veure que per a fer aquesta consulta, caldrà mirar tots els valors de la taula Persona i mirar

la seva edat per a saber si compleix la condició. Això implica que si la taula és gran, la consulta serà costosa. Afegint un simple índex (un arbre B+ per exemple) sobre la columna edat, aquest temps de consulta passaria a de ser $O(n)$ a $O(log\ n)$.

Existeixen diferents tipus d'índexs però els més comuns son el de Hash i els arbres B+.

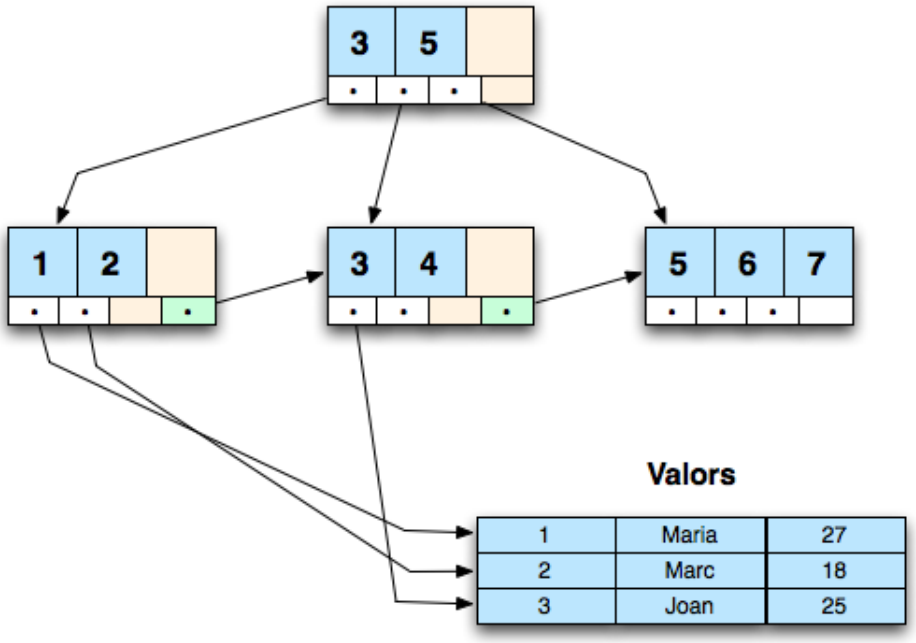
Índex de Hash Aquest índex el que fa és codificar el valor sobre el que construïm l'índex i desar-lo a una estructura de dades juntament amb la referència al seu valor original. Aquest tipus d'índex ens serveix per a fer cerques de valors concrets, i ens ofereix un cost d'$O(1)$.



Listing 6: Índex de Hash

Com es pot veure a la figura 6, s'agafa la clau sobre la que volem construir l'índex, en aquest cas el nom, aquest valor passa per una funció de Hash i s'en treu una clau associada. Aleshores a l'estructura de dades de l'índex es desa aquesta clau, i l'enllaç cap a la fila de la base de dades que conté el registre.

Arbres B+ Aquest tipus d'índex, construeix un arbre balancejat del tipus B+ sobre el valor desitjat. Aquest tipus d'índex ens serveix per a optimizar cerques de rangs de valors, el cost de cerca d'un rang de valors passa a ser $O(log\ n + k)$ on k és el nombre de valors que busquem. El cost de cerca per a un valor concret també millora, passa a ser $O(log\ n)$.



Listing 7: Índex d'arbre B+

A la figura lst:intro-b+ tenim un exemple d'índex amb arbre B+. En aquest exemple l'índex es construeix sobre l'id de la fila. Amb els id's es construeix l'arbre, i a les fulles de l'arbre tenim la referència cap a la fila de la base de dades que conté el registre.

Joins

Les joins son una operació del llenguatge SQL, però és una operació clau. És la que dona la màxima potència a aquest llenguatge. Les joins ens permeten fer consultes a vàries taules a la vegada. Normalment es volen unir vàries taules usant les claus que les relacionen.

```
SELECT p.nom, p.edat, c.nom as ciutat FROM persona p, ciutat c
    WHERE p.idCiutat=c.id;
```

Listing 8: Consulta per a la obtenció de dades en SQL

Nom	Edat	Ciutat
Joan	25	Barcelona
Maria	27	Madrid

Listing 9: Resultat de la consulta

Amb la consulta de la figura 8, obtindriem un resultat com de la figura 9. I veiem com usem les dues taules usant la clau que les relaciona, l'id de la ciutat.

2.1.2 Limitacions dels SGBDR's

Ara que ja hem fet un repàs a les característiques principals dels SGBDR's , anem a veure com aquestes característiques i aquesta flexibilitat que ens aporten, comporten a la vegada limitacions. Tot i que aquestes limitacions, normalment son imperceptibles i només en comencem a prendre consciència quan portem els sistemes fora dels seus entorns habituals, és a dir, quan forcem els seus límits.

Model de dades

Els esquemes de les taules dels SGBDR's, és considerat estàtic. Això vol dir que una vegada tenim una taula amb registres, canviar aquest esquema no és trivial.

Cada taula està composta per columnes, i totes les files d'una taula, tenen un valor associat per a cada columna d'aquesta. Això te dues conseqüències importants.

nulls Totes les files d'una taula han de tenir un valor per totes i cadascuna de les columnes, i moltes vegades una columna a una fila no tindrà cap sentit i la voldrem deixar buida. L'equivalent al valor buit en SQL és el valor null. Els *nulls* tenen dos contrapartides, ocupen espai innecessari a disc, i introdueixen problemes de rendiment (i de comportament si no es van en compte) a les cerques.

Reconstrucció de taules Si volem realitzar qualsevol canvi a l'esquema hem de re-definir l'esquema. Qualsevol canvi a l'esquema, normalment el que

comporta és haver de reconstruir totalment la taula, per a fer això el que es fa normalment, a nivell intern del SGBD, és crear una taula nova amb la nova estructura, copiar tots els valors a aquesta nova taula, i afegir el valor per defecte a les noves columnes que s'hagin afegit a l'esquema.

Aquests canvis d'esquema en taules de mida moderada, no és un problema massa gran. Però quan estem tractant amb taules amb milions de registres, normalment son tan costosos que poden trigar hores o inclús dies.

Índexs

Els índexs son un element essencial dels sistemes relacionals, els beneficis dels índexs son tan elevats que el més habitual és que qualsevol taula del nostre model de dades, tingui múltiples índexs.

Els índex a nivell intern del SGBD son estructures de dades mantingudes per separat de la taula. Cada vegada que fem una operació a una fila, s'ha de reflectir als índexs que afectin aquesta fila. Això implica que si per exemple tenim un índex de hash, al inserir una fila a una taula, hem de calcular el valor de hash d'aquella fila i inserir-lo a l'estructura de dades que emmagatzema l'índex en qüestió.

Així que els índexs optimitzen les cerques però empitjoren les insercions, ja que per a cada fila que inserim /editem /eliminem a una taula, haurem de fer les operacions indicades als índexs corresponents. Els índexs també impliquen un cost extra en espai, ja que les estructures de dades s'han de desar a disc, tot i que a mida dels índexs acostuma a ser bastant més petita que la de les dades en si.

A part d'aquests costs, a l'operació on realment penalitzen els índexs es a l'hora de reconstruir les taules. Com hem vist a l'apartat anterior, al modificar l'esquema d'una base de dades hem de reconstruir la taula. Si la taula té índexs, aquests s'hauran de reconstruir. Per tant reconstruir una taula passa a ser encara més costós quan aquesta té índexs.

Joins

Les *joins*, com hem vist son una operació molt potent del SQL, però a la vegada son la seva eina més perillosa. Si fem consultes de múltiples taules, necessitem fer-les de tal manera que aquestes operacions d'unió no siguin massa costoses. Si agafem per exemple la consulta de la figura 8, imaginem que passaria si la taula ciutat no tingués cap índex definit. Per cada fila de la taula Persona, hauríem d'agafar el seu idCiutat, aleshores buscar a la taula ciutat aquest id i ajuntar les files. Com que hem dit que la taula ciutat no té cap índex, buscar un id té un cost $O(n)$.

Si agafem aquest petit cas i ho extrapolem a consultes amb més d'una *join*, i amb taules amb milers o milions de files, podem veure que aquesta és una operació perillosa, ja que podem crear consultes molt lentes.

Aquest problema que suposen les *joins*, es pot evitar amb un bon disseny de la base de dades , posant totes les clau necessàries per a optimitzar aquestes consultes, però com hem vist, posar índexs no és sempre una bona idea.

2.1.3 Necessitats dels entorns d'alta demanda

En qualsevol aplicació, el que es busca és que l'aplicació funcioni. Així de simple. Podríem dir que l'objectiu principal de qualsevol aplicació informàtica és (o hauria de ser) la usabilitat. Per a que una aplicació sigui usable una de les característiques més importants és la velocitat de resposta, volem que tots els usuaris que usen una aplicació, tinguin temps de resposta acceptables, i encara més si parlem d'entorns web. Als entorns web, es considera que una aplicació deixa de ser usable quan els seus temps de resposta superen els 3 segons, i la recomanació és que tots els temps de resposta estiguin al voltant del segon.

Per a que una aplicació sigui sempre usable, es diu que l'aplicació (i la seva infraestructura) ha de ser escalable, això vol dir que l'aplicació tingui la capacitat d'adaptar-se a la demanda que pugui tenir en qualsevol moment, tant si parlem de 10 usuaris com de 10.000.000. El més habitual es que quan es dissenya una aplicació no se sàpiga quin serà el seu abast, per tant es dissenya pensant en entorns de demanda moderada, el problema ve quan les previsions es superen i s'arriba a tenir quotes de demanda que no s'havien previst. En aquest moment és quan és important l'escalabilitat, per poder reaccionar a aquests increments de demanda fàcilment. I com veurem més endavant, escalar els SGBDR's no és una tasca fàcil.

A part de l'escalabilitat, una altra característica molt important és l'alta disponibilitat, és a dir, que en cas de que hi hagi problemes com caiguda de servidors, l'aplicació segueixi funcionant.

2.1.4 Entorns d'alta demanada amb sistemes relacionals

Amb aquesta introducció a les necessitats dels entorns d'alta demanda i amb la descripció dels sistemes relacionals, anem a veure quines eines posen a la nostra disposició els SGBDR's per a poder construir entorns d'alta demanda.

En aquest apartat es farà servir com a referent MySQL ja que és el SGBDR de codi lliure estàndard del mercat actual.

Alta disponibilitat

L'alta disponibilitat de les dades fa referència al fet de tenir vàries còpies de les dades, per si hi hagués algun problema en un servidor (o *datacenter*) poder obtenir les dades d'alguna de les còpies disponibles.

Això també ens serveix per si tenim varis *datacenters*, poder tenir aquests *datacenters* ubicats en llocs estratègics per tal de que estiguin aprop dels usuaris que necessiten les dades. Avui en dia, les grans aplicacions d'Internet son globals i tenen usuaris per tot el món, això fa que tenir varis *datacenters*, pugui contribuir a tenir millors temps de resposta.

La manera d'obtenir aquesta alta disponibilitat és mitjançant la replicació de les dades.

Replicació

Com acabem de veure l'alta disponibilitat és una necessitat en molts casos, i la manera d'aconseguir-la és mitjançant la replicació.

Als SGBD's actuals, aquesta replicació s'aconsegueix fent còpies complertes de cada node. El que s'ofereix és tenir nodes màster i nodes esclau, un node esclau és una còpia exacte d'un node màster. Els nodes esclaus, el que fan és anar reproduint el dietari del node màster.

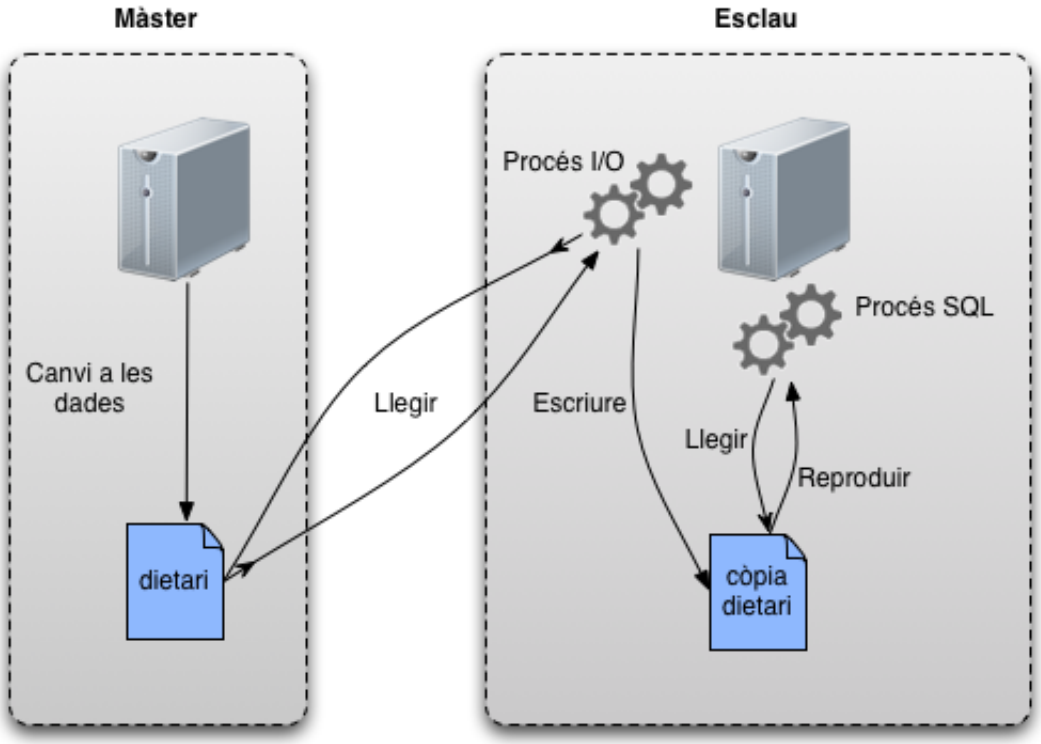
Amb aquests dos elements podem tenir moltes topologies diferents. Amb aquest tipus de replicació tenim varis avantatges:

- Redundància de les dades
- Escalabilitat de les lectures
- Distribució geogràfica

Però també tenim certes limitacions:

- Únic punt de fallida (màster)
- No escalabilitat de les escriptures
- Complexitat de manteniment

Amb aquesta llista de limitacions, cal notar que 2 de les 3, venen donades per la topologia màster-esclau. Al necessitar un màster, directament veiem que si aquest deixa de funcionar, també ho faran tots els seus esclaus. I a part, com podem veure a la figura 10, els esclaus llegeixen el dietari del màster, això implica que per a fer una escriptura, forçosament s'ha de fer



Listing 10: Estructura màster-esclau

primer al màster i després als esclaus. Això implica que les escriptures no son escalables, ja que només es realitzen a un dels nodes. No passa el mateix amb les exriptures, ja que una vegada un esclau té les dades, aquestes es poden llegir del node esclau.

Escalabilitat

Com hem vist a l'apartat anterior, la replicació ens ofereix escalabilitat de lectures, però no d'escriptures, per tant si volem tenir un sistema realment escalable hem de mirar altres maneres d'obtenir-la. Hi ha diferents tipus d'escalabilitat, però les més habituals son dues:

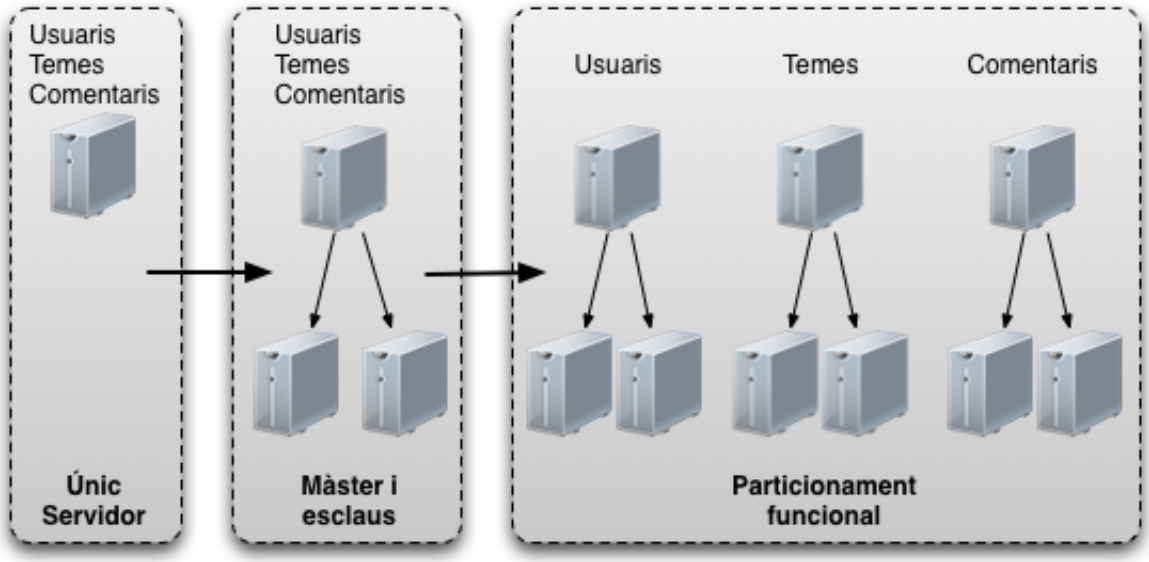
- Escalabilitat vertical: Consisteix en augmentar els recursos del servidor. Aquests augments tenen un límit marcat pel hardware disponible i per tant no és el model a seguir, tot i que pot ajudar a sortir del pas en determinades ocasions.

- Escalabilitat horitzontal: Consisteix en dividir la feina en diversos servidors, aquest model és l'utilitzat habitualment, ja que és el que realment aconsegueix obtenir resultats òptims.

La manera d'obtenir escalabilitat horitzontal als SGBDR's, és mitjançant el particionament (sharding). El que es fa habitualment per a obtenir aquest particionament és separar el sistema en varis subsistemes independents, i posar cada un d'aquests subsistemes en un node separat.

Aquest model, no només té la dificultat de configuració i manteniment dels servidors, sinó que és molt complicat fer un bon particionament en subsistemes totalment independents, i habitualment, necessitarem fer joins amb dades de diferents nodes en algun punt de l'aplicació, i això s'ha de fer a nivell d'aplicació i no de SGBD, i aquí és on es compliquen realment les coses.

A part d'aquesta complicació s'ha de sumar que aquests diferents subsistemes també han de ser replicats per ser altament disponibles. Per tant s'ajunten moltes coses, que fan que mantenir un sistema realment escalable i altament disponible, requereixi d'experts en la matèria i sigui una feina molt farragosa.



Listing 11: Possible particionat de taules

A la figura 11 veiem les possibilitats per a escalar un sistema MySQL. Primer posem dos nodes esclaus que ens serveixen per a tenir replicació de

les dades i de pas escalem les lectures. Però quan necessitem més, entra en joc el particionament. El que fem és separar el sistema en 3 subsistemes, un per a cada taula del sistema original. Amb aquest particionament tenim que hem escalat les escriptures, ja que ara cada taula pertany a un únic màster, i també escalem les lectures ja que cada taula té també dos nodes esclau. El gran problema del particionament és que si necessitem fer joins entre les taules, al estar cada una a un sistema diferent, és molt més complexe ja que no podem fer una simple consulta join.

Com veurem als pròxims apartats, els nous sistemes que estudiarem, son sistemes on el seu model de dades és directament el resultat de particionar els models de dades.

2.1.5 Aparició de noves eines

Hem fet un repàs a les principals característiques dels SGBD's relacionals referents als entorns d'alta demanda. I queda clar que tot i que son capaços d'adaptar-se a aquests entorns, i cada vegada incorporaran noves eines per a fer-ho, la seva estructura base no està dissenyada per a fer front a aquests entorns. Això fa que cadascuna de les eines que ens ofereixen, siguin difícils de configurar i mantenir, tenint sistemes moltes vegades una mica inestables i que requereixen de molta dedicació. I en un projecte gran amb molt de pressupost tenir gent dedicada a mantenir aquests sistemes, no és un problema. Però en aplicacions emergents, seria molt més útil i eficient, dedicar tots els esforços a dissenyar l'aplicació i no en mantenir les bases de dades.

Una de les empreses que va començar a tractar amb volums de dades tant grans que començaven a tenir problemes amb les bases de dades convencionals va ser Google. Al 2006, Google va publicar un paper anomenat [1] . A aquest paper, Google proposa un nou model per emmagatzemar dades basat en les seves necessitats . A partir d'aquí va néixer el que avui es coneix com a NoSQL.

2.2 Sistemes orientats a columnes

2.2.1 Què és NoSQL

El terme NoSQL [4] és un terme que va sorgir al 1998 amb una petita base de dades, creada per Carlo Strozzi, que no oferia una interfície d'SQL. El nom va tornar a aparèixer al 2009 quan Eric Evans, treballador de Rackspace, va voler donar nom a una conferència organitzada per parlar sobre bases de

[4]Not Only SQL

dades distribuïdes de codi lliure. Com l'autor original del nom va dir, en realitat la paraula no és adequada ja que el que busquen aquests sistemes és separar-se del model relacional, per tant una paraula més adequada hagués estat NoREL. Tot i que NoSQL no és el nom més indicat, ha esdevingut el nom estàndard per a aquests sistemes i per tant és el que es farà servir en aquest document per fer-hi referència.

El món de NoSQL engloba tot tipus de bases de dades distribuïdes no relacionals, nosaltres ens centrarem en les orientades a columnes. Per tant a partir d'ara sempre que parlem de NoSQL estarem fent referència a sistemes de bases de dades distribuïts orientats a columnes.

2.2.2 Naixement i estat actual

Com hem vist, el naixement dels sistemes NoSQL ve marcat per la necessitat d'emmagatzemar grans quantitats d'informació en entorns distribuïts. Les limitacions (o complexitat) dels sistemes relacionals al respecte, van portar a Google a desenvolupar un nou model de base de dades. Sembla ser que van començar a usar BigTable al 2004, tot i que el portaven desenvolupant des de bastant temps enrere, però no va ser fins al 2006 quan van publicar [1]. A partir d'aquí van començar a aparèixer diferents implementacions de codi lliure basades en aquest model.

Actualment les diferents plataformes NoSQL que existeixen, es troben encara en fase de desenvolupament, totes en versions *beta*. La majoria de projectes compten amb una bona base de desenvolupadors i se'n treuen versions noves contínuament. Si considerem que es coneix de varis casos d'empreses que al 2008 ja tenien clústers amb algun Software NoSQL en producció, ens adonem de la necessitat que tenien les empreses de trobar sistemes de bases de dades distribuïts eficients. Cada dia son més les empreses que usen solucions NoSQL dins de les seves aplicacions, tot i que la majora d'elles sols les usen per a certes parts de les seves aplicacions. Gairebé totes les grans empreses d'Internet actuals, usen NoSQL. Empreses tals com Google, Facebook, Digg, Twitter, etc.

El fet de que empreses tan importants confiïn en aquests sistemes, son la major garantia de que els sistemes son estables i funcionals. Sobretot ens fa veure que si son capaces d'arriscar-se a usar Software en fase beta, es que la contrapartida d'haver de configurar un SGBDR és molt pitjor que arriscar-se a usar Software en versió beta.

2.2.3 Usos principals

Tot i que durant tot el document estarem fent comparacions entre NoSQL i els SGBDR's, cal tenir molt clar que l'objectiu dels sistemes NoSQL no és substituir els SGBDR's. Cadascun dels dos té usos molt diferents i cal saber reconèixer quan és realment necessari usar un sistema NoSQL i quan no. Totes les comparacions que es facin en aquest document, son per posar al lector en context respecte al que ja coneix, els sistemes relacionals.

L'objectiu dels sistemes NoSQL és oferir un sistema de bases de dades distribuïda per a entorns d'alta demanda. Quan parlem d'entorns d'alta demanda, com ja hem dit, parlem d'entorns amb milers o milions d'usuaris, i arribant fins a PetaBytes de dades. Per tant això ja limita molt els casos en que els sistemes NoSQL son útils. Qualsevol empresa o persona que no tingui problemes de coll d'ampolla amb el seu SGBD actual, i mai s'hagi plantejat haver de fer replicació, particionament etc. de les seves taules, el més provable és que els sistemes NoSQL no siguin per a ell. Tampoc son adequats els sistemes NoSQL als entorns a on hàgim d'aprofitar molt sovint els avantatges dels SGBD's relacionals com la indexació, o les consultes complexes mitjançant múltiples *joins*, etc.

Normalment el que es diu es que si mai has tingut mal de cap amb temes de particionament (*sharding*), distribució, etc. amb el teu SGBD actual, el més provable és que no necessitis un sistema NoSQL.

Per tant, ha de quedar clar, que els sistemes NoSQL s'han d'usar en entorns d'alta demanda, on realment és necessari tenir un SGBD distribuït.

2.2.4 Característiques dels sistemes NoSQL

Com hem dit hi ha vàries implementacions de sistemes NoSQL, i cadascuna d'elles té característiques que la diferencien. Dins de tots els sistemes existents, per exemple, n'hi ha enfocats a consultes OLAP [5], i d'altres cap a OLTP [6]. Aquesta és una diferenciació entre alguns sistemes, tot i que n'hi ha d'altres.

És difícil fer una caracterització genèrica dels sistemes NoSQL, així que per a realitzar aquest projecte el que farem serà escollir-ne dos i analitzar-los. Segons el criteri exposat anteriorment, hem triat:

- HBase (OLAP)
- Cassandra (OLTP)

[5] Online Analytical Processing
[6] Online Transaction Processing

A part de la diferència entre OLAP i OLTP, veurem que amb un sol punt de partida (BigTable) es poden crear sistemes totalment diferents.

Al següents capítols explorarem aquests dos sistemes, però primer aprofitarem els següents apartats per introduir uns quants conceptes que ens seran útils per a parlar d'aquests dos sistemes.

De-normalització dels esquemes

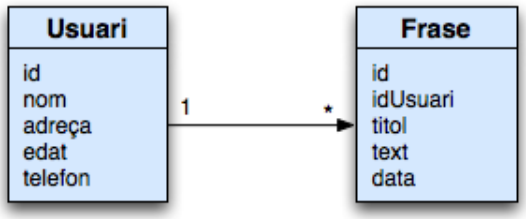
La gran diferència respecte els SGBDR's, la trobem a l'hora de dissenyar els esquemes de les bases de dades. A un SGBDR, podem definir mitjançant índexs i claus foranies, relacions entre les dades. I a l'hora d'obtenir les dades, podem executar consultes complexes usant operadors com les *join*, que ens permeten fer consultes complexes i obtenir una gran quantitat de dades usant sempre les mateixes taules amb el mateix disseny.

Això no passa als sistemes NoSQL. Els sistemes NoSQL tenen un model de dades diferent, una fila no pertany a una taula sinó a una família de columnes (*Column Family)*, i totes les files s'identifiquen per una única clau (índex de la fila). Al tenir un únic índex, les consultes que podem realitzar son simplement obtenir una sèrie de valors basant-nos en una clau. És a dir, tenim la clau d'una fila i podem obtenir o bé tota la fila, o només un rang de columnes. Però aquesta és la màxima flexibilitat que tenim. Alguns sistemes ens permeten fer *scans* de vàries files o bé *scans* dels valors de dins d'una columna, però sempre donant una o vàries claus que hem de conèixer amb antelació.

Aquesta estructura, ens porta directament a la de-normalització dels esquemes. Amb un esquema de-normalitzat el que fem és crear les taules pensant en les consultes que volem realitzar, no creem models amb UML i després creem les taules corresponents a les diferents entitats del sistema.

Un dels grans problemes de tenir les dades de-normalitzades, és la repetició de les dades. És molt provable que tinguem les mateixes dades múltiples vegades en diferents taules, ja que al fer les taules pensant en les consultes, pot ser que a vàries consultes necessitem una mateixa dada. Per exemple, al agafar missatges d'un sistema de missatgeria, necessitem informació sobre l'usuari que ha enviat el missatge (nom, id, etc), però aquesta informació també estarà a la taula d'usuaris perquè també ens cal al buscar els usuaris. Això fa que s'hagi d'anar amb molta cura per mantenir la coherència de les dades.

Anem a veure un petit exemple de com crearíem un petit sistema amb les dades normalitzades i de-normalitzades. Imaginem que tenim un sistema que permet als usuaris escriure missatges. Es a dir, tenim dues entitats, usuaris i frases.



Listing 12: Esquema UML

Ara imaginem que tenim una web amb dues vistes, a una podem veure tots els missatges que ha escrit un usuari, i a l'altre podem veure tots els missatges que s'han inserit al sistema, en ordre cronològic.

Aquest exemple amb un SGBDR, constaria de dues taules, una d'usuaris i una altra de missatges (amb clau forània sobre l'id de l'usuari que ha escrit el missatge) tal i com es mostra a la figura 12, i faríem les dues consultes mitjançant SQL sobre aquestes dues taules.

Les taules quedarien així:

Usuaris

idUsuari	nom
1	Joan
2	Maria

Missatges

idMissatge	idUsuari	timestamp	missatge
1	1	t1	"Soc en Joan"
2	2	t2	"Soc la Maria"
3	1	t3	"M'agrada NoSQL"

Listing 13: Taula d'usuaris i missatges

I les dues consultes serien així:

```
Select u.nom, m.timestamp, m.missatge From Usuaris u, Missatges m
    Where u.id=m.idUsuari Order By timestamp desc
Select u.nom, m.timestamp, m.missatge From Usuaris u, Missatges m
    Where u.id=m.idUsuari and u.id='1' Order By timestamp desc
```

Listing 14: Conslutes en SQL

Si ara volem fer el mateix sobre un sistema NoSQl, tenim múltiples opcions, però per al cas que ens ocupa que és mostrar la de-normalització de les dades, en mostrarem dues i els seus diferents avantatges o desavantatges. Per als exemples no seguirem cap notació estricte, ja que encara no hem parlat de com es creen els esquemes als sistemes NoSQL, i el que ens interessa és únicament exposar un exemple.

La primera temptació seria mantenir les taules d'usuari i missatges, però ens falta un mètode per a desar les relacions entre missatges i usuaris. El que podem fer és afegir una columna a la taula d'usuaris que contingui els id's de tots els missatges que ha escrit aquell usuari. Amb aquesta estructura tindríem una cosa similar a això:

Usuaris

idUsuari	nom	missatges
1	Joan	1,3
2	Maria	2

Missatges

idMissatge	idUsuari	timestamp	missatge
1	1	t1	"Soc en Joan"
2	2	t2	"Soc la Maria"
3	1	t3	"M'agrada NoSQL"

Listing 15: Primera versió d'usuaris i missatges denormalitzats

Per a obtenir tots els missatges d'un usuari, hem d'anar a la taula d'usuaris, agafar el contingut de la columna missatges i després anar a la taula de missatges a buscar tots els missatges. Per agafar els missatges per ordre

cronològic hauríem de fer la ordenació al client, o bé triar una altra clau primària.

A part de que l'esquema no s'adapta bé a les consultes que volem realitzar, també hem de tenir en compte que desar els id's dels missatges a una cel·la de la taula d'usuaris, pot provocar problemes d'espai. Per a solucionar aquest problema, el millor que podem fer és tenir una taula per a desar aquestes relacions, per exemple una taula anomenada missatges-usuari, on la clau de la fila sigui l'id de l'usuari i el contingut el valor del missatge. A priori sembla una mala solució ja que només podrem tenir una fila per usuari, però això ho podem solucionar creant una clau composta amb l'id de l'usuari i el *timestamp* del moment en que s'ha escrit el missatge. Ens centrarem molt més en construcció d'índexs en el següent capítol.

Un altre problema important és el fet de que per cada id de missatge escrit haurem de fer una consulta a la taula de missatges. Amb lo qual haurem de fer un nombre elevat de consultes. Això ho podem resoldre si implementem la taula missatges-usuari, i hi afegim una columna amb el contingut del missatge.

Si agafem aquestes dues modificacions i polim una mica més el sistema, obtenim les següents 3 taules:

Usuaris

idUsuari	nom
1	Joan
2	Maria

Missatges-Usuari

idUsuari + timestamp	missatge
1+t1	"Soc en Joan"
1+t3	"M'agrada NoSQL"
2+t2	"Soc la Maria"

Missatges-Timestamp

timestamp	missatge
t1	"Soc en Joan"
t2	"Soc la Maria"
t3	"M'agrada NoSQL"

Listing 16: Segona versió d'usuaris i missatges de-normalitzats

Per agafar els missatges d'un usuari hem d'anar a la taula missatges-usuari i agafar les files que comencen amb l'id de l'usuari que volem. Mentre que per a agafar els missatges en ordre cronològic anem a la taula missatges-timestamp i agafem les primers files que contingui.

Segurament sorgeixen bastants dubtes respecte a aquesta solució, i veurem que les respostes depenen del sistema NoSQL que triem. Per exemple, podem veure que s'hauran de fer tantes consultes com missatges vulguem obtenir, mes una per a cada usuari que vulguem obtenir, per tant, no hem millorat gens respecte la solució anterior. Doncs no és cert, ja que com veurem, el fet de tenir les files agrupades de la manera com ho estan, farà que puguem obtenir tots els missatges en una única consulta. I el tema d'haver de fer una consulta per usuari, ho podríem solucionar desant la informació de l'usuari que ens fes falta a les dues taules de missatges com es pot veure a la figura 17, això ens porta al següent problema.

Usuaris

idUsuari	nom
1	Joan
2	Maria

Missatges-Usuari

idUsuari + timestamp	missatge	nomUsuari
1+t1	"Soc en Joan"	Joan
1+t3	"M'agrada NoSQL"	Joan
2+t2	"Soc la Maria"	Maria

Missatges-Timestamp

timestamp	missatge	nomUsuari
t1	"Soc en Joan"	Joan
t2	"Soc la Maria"	Maria
t3	"M'agrada NoSQL"	Joan

Listing 17: Modificació de la segona versió d'usuaris i missatges de-normalitzats

El fet de desar tanta informació repetida és el problema més clar de de-normalitzar els esquemes, i aquest problema no té una solució clara. En la majoria de casos ens hem de plantejar si poder obtenir les dades que volem en una sola consulta, amb el guany en velocitat de resposta que comporta, ens importa suficientment com per a haver de mantenir totes les còpies de la informació manualment al nostre codi.

Com hem vist abans, els sistemes NoSQL no son per a tothom ni s'adapten a tots els usos, i el fet d'haver de de-normalitzar els esquemes, normalment fa enrere a molta gent. Normalment el punt en que algú decideix passar a sistemes NoSQL és quan han de començar a de-normalitzar les seves taules sobre un SGBD relacional per temes d'eficiència i de particionament (*sharding*).

Localització de les dades

Un aspecte molt important als sistemes NoSQL i que va molt lligat a la normalització de les dades, és el de la localitat de les dades. A l'hora de desar les dades s'ha de decidir a quin node es deixen les dades, i això té molta importància als sistemes NoSQL.

L'assignació de les dades es fa mitjançant els particionadors. Normalment el particionador és un element que es defineix globalment a nivell de clúster. Aquests particionadors son mecanismes molt senzills que tenen en compte dos elements:

- Rang de valors associat a cada node
- Clau de les files

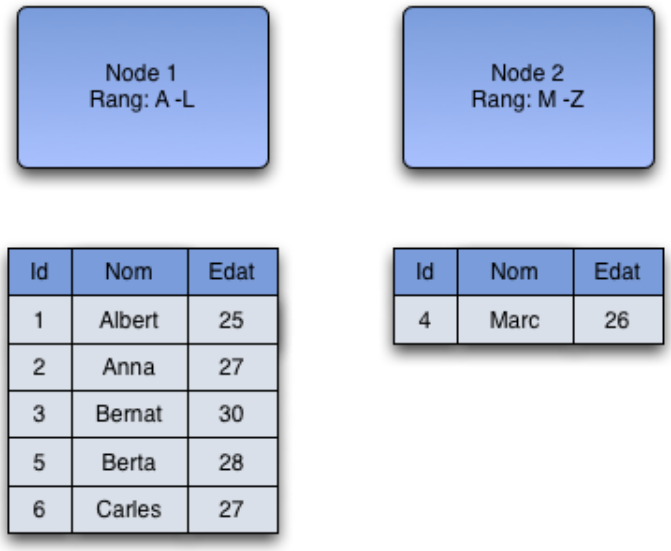
Cada node té assignat un rang de valors, aleshores al inserir una nova fila s'agafa la seva clau i es mira el rang de cada node. La fila s'insereix al node que té assignat el rang al que pertany la clau de la fila. Els diferents particionadors tenen diferents estratègies per assignar els rangs als nodes i per a tractar les claus de les files. Als pròxims apartats veurem les estratègies que fan servir alguns dels particionadors més comuns.

El funcionament òptim de qualsevol sistema NoSQL s'aconsegueix quan un clúster està ben balancejat, és a dir, quan la càrrega de cada node és el més homogènia possible. És tasca del desenvolupador triar el millor particionador i assignar els rangs el millor possible per a aconseguir tenir la repartició de dades òptima. Com veurem hi ha diferents particionadors i cadascun és més adient per a diferents situacions.

Veiem un exemple de com un mateix conjunt de dades amb diferents claus i particionadors afectarien a la repartició de dades entre nodes.

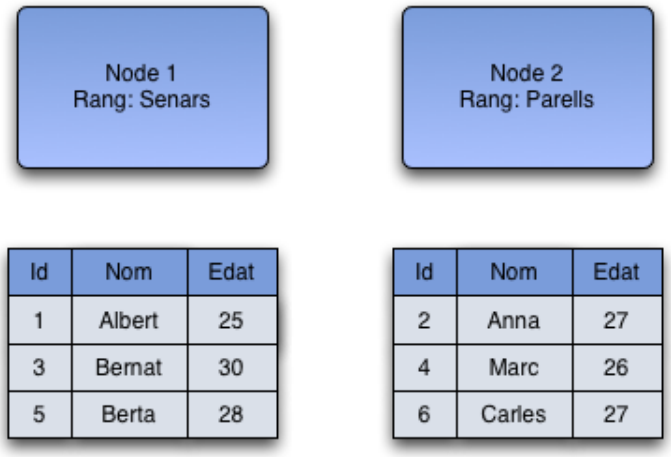
Id	Nom	Edat
1	Albert	25
2	Anna	27
3	Bernat	30
4	Marc	26
5	Berta	28
6	Carles	27

Listing 18: Conjunt de dades



Listing 19: Particionador lèxic

A la figura 18 definim el conjunt de dades que contindrà el clúster de l'exemple. A la figura 19 veiem com quedarien repartides les dades si féssim servir un particionador que repartís les dades segons ordre alfabètic i assignant a cada node la meitat de l'alfabet. Com veiem queda una repartició totalment descompensada. Teòricament aquest és un cas excepcional ja que amb un *set* de dades major, la repartició hauria de ser molt més homogènia, però ens serveix per a veure com, si seleccionem un particionador no adequat per al *set* de dades que tindrem, això pot afectar negativament al clúster. En aquest cas en concret, el node 1 rep moltes més peticions d'escriptura i conseqüentment rebrà moltes més peticions de lectura i modificació, mentres que el node 2 tindrà molt menys càrrega proporcionalment.



Listing 20: Particionador parells-senars

A la figura 20 veiem com agafant el mateix set de dades però fent la partició sobre l'id de la fila, mirant si aquest és parell o senar, obtindrem una repartició totalment homogènia. Aquest és el que es considera (en termes de particionament) un clúster ideal, ja que està totalment balancejat.

Un altre aspecte que s'ha de tenir en compte a l'hora de triar el particionador, i per tant, decidir la localitat de les dades, és el tipus de consultes que voldrem realitzar. Normalment al fer una consulta voldrem seleccionar múltiples files. Al seleccionar múltiples files, com amb qualsevol sistema, ens interessa fer el mínim d'operacions de consulta sobre el disc dur. Per tant, el millor serà tenir totes les dades que volem seleccionar ben agrupades al disc dur. Així, si agafem per exemple l'exemple de la figura 19, i volem fer una cerca que agafi totes les persones que el seu nom comenci per "A", presumiblement les dades estaran consecutives al mateix bloc del disc dur, com a mínim podem garantir que es trobaran al mateix clúster. Mentre que amb l'exemple de la figura 20 la mateixa consulta hauria de recollir dades dels dos nodes, això implica que la mateixa consulta haurà de fer consultes a dos servidors diferents que al seu temps hauran de buscar les dades als seus respectius discs durs.

Amb aquests petits exemples veiem com la decisió a l'hora de triar el particionador del clúster és molt important i va directament lligada al model de dades que fem servir i les claus que decidim fer servir per a cada taula.

Cada sistema NoSQL té els seus mètodes de particionament i d'assignació de rangs als nodes, però a gairebé tots els sistemes en trobem dos de comuns

que analitzarem a continuació:

- Particionador lexicogràfic
- Particionador aleatòri

Particionador lexicogràfic Aquest particionador no efectua cap operació sobre la clau. Simplement fa una ordenació lexicogràfica de les claus tal i com les rep. Aquest particionador no ens ofereix cap tipus de garantia respecte a la distribució de les claus, per tant s'ha d'anar amb compte en fer-lo servir.

La ordenació lexicogràfica és realment útil si es volen obtenir rangs de claus ordenades. Per exemple, si tenim un sistema on desem els missatges d'un usuari, de tal manera que la clau sempre serà "nomUsuari+timeStamp" i serà molt habitual fer cerques de més d'un missatge a la vegada, com per exemple els 10 últims missatges enviats, aquest tipus de particionador ens serà realment útil. Ja que el més habitual serà que tots els missatges es trobin al mateix node i no només això sinó que en posicions consecutives de disc, per tant amb una sola operació de lectura sobre el disc obtindrem tots els missatges. Mentre que si féssim servir el particionador aleatori, el més provable seria que haguéssim d'obtenir els missatges de diferents nodes i diferents regions de disc, per tant el temps de consulta seria molt més elevat.

El gran desavantatge del particionador lexicogràfic és que no tenim cap garantia respecte a la distribució de les dades, hem de preocupar-nos nosaltres de proporcionar aquestes garanties, ja sigui o bé usant claus ben pensades per a obtenir aquesta distribució, o bé triant manualment el rang de claus de cada node. En l'exemple anterior, si repartíssim un nombre igual de lletres de l'alfabet a cada node, però tots els usuaris triessin un nom d'usuari començat en "a", tota la càrrega del sistema aniria parar a un mateix node.

La ordenació lexicogràfica és la que fa servir HBase, mentre que Cassandra fa servir per defecte el particionament aleatori, com és propi de les taules de Hash distribuïdes. No es recomana fer servir el particionador lexicogràfic a no ser que realment sapiguem que és el que necessitem i les nostres dades s'hi adapten perfectament (tindran una bona distribució).

Particionador aleatori El particionador aleatori, el que fa és generar una clau "aleatoria"basada en la clau que que tingui la fila. Aquesta clau es genera amb algun d'algorisme [7], i una vegada generada aquesta clau s'assigna al node que correspongui. Als nodes se'ls assigna un rang de claus equitatiu. S'agafa el rang de claus generable per l'algorisme i es divideix entre el nombre de nodes disponibles.

[7] MD5 en el cas de Cassandra

Aquesta estratègia de particionament acostuma a donar molts bons resultats, i no depèn del *set* de dades ni requereix cap acció per part del desenvolupador. Però el seu funcionament fa que el desenvolupador no tingui cap tipus de control sobre la localitat de les dades. Trii les claus que trii, la localitat de les dades depèn del resultat de la codificació de les claus. Això fa que si necessitarem fer consultes de rangs, no sigui el particionador adequat.

Aquest particionador és el que ve activat per defecte a Cassandra.

CAP-Aware

Hi ha un teorema, anomenat teorema de CAP, formulat per Eric Brewer, que ens diu que hi ha 3 característiques fonamentals per a qualsevol sistema distribuït:

- Consistència (***C****onsistency*)
- Disponibilitat (***A****vailability*)
- Tolerància al particionament de la xarxa (*Tolerance to* ***P****artitions*)

D'aquestes característiques, qualsevol sistema distribuït que vulgui oferir una latència acceptable, només en pot oferir 2. Això com veurem es compleix amb els sistemes que analitzarem i de fet és el que fa que els sistemes puguin ser diferents i adequats per a usos diferents.

Consistència La consistència implica que el contingut de la base de dades sempre és consistent. Això vol dir que si s'escriu un valor a la base de dades, qualsevol lectura feta després d'aquesta escriptura, obtindrà aquest el valor escrit. Si agafem per exemple un compte corrent, i fem una operació que passa el saldo de 50 a 25, la pròxima vegada que consultem el saldo ens ha de dir que disposem de 25 no 50.

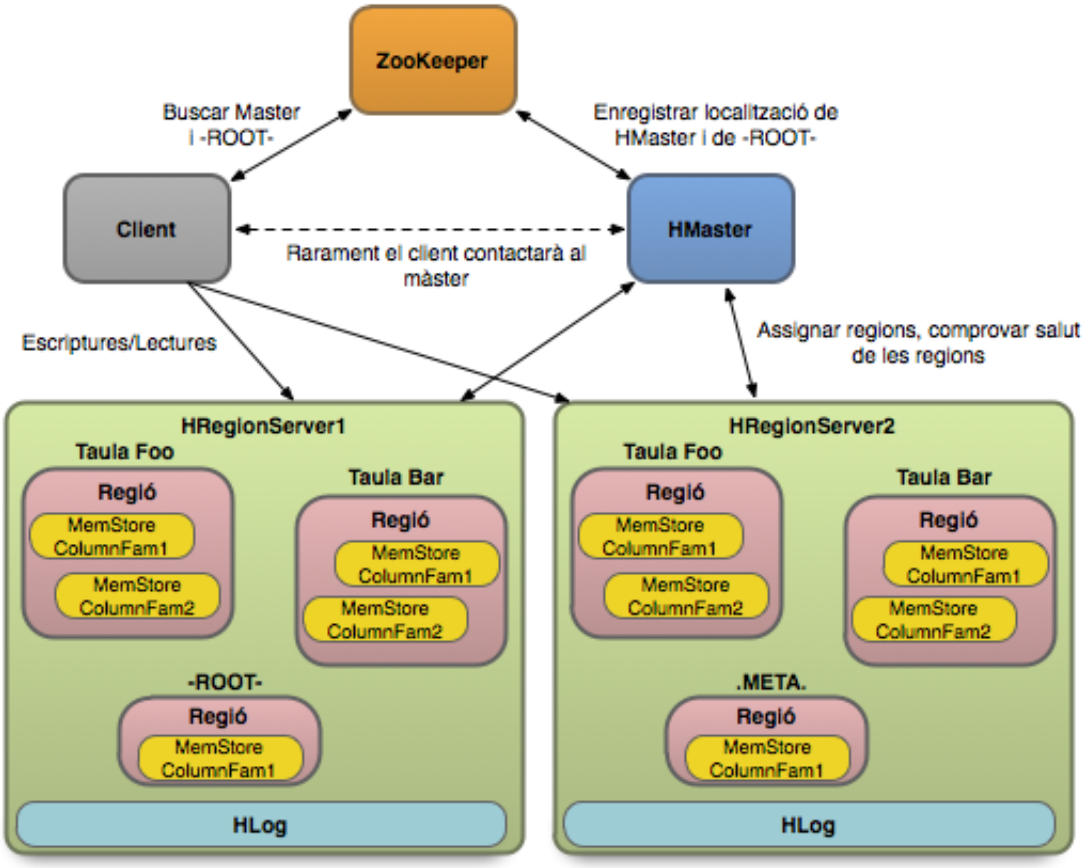
Disponibilitat La disponibilitat és el que ens indica la fiabilitat del sistema. Un sistema amb alta disponibilitat és un sistema on poden çaure" molts nodes sense afectar al sistema. Mentre que un sistema a on si çau" un node el sistema deixa de funcionar correctament, és un sistema amb molt baixa disponibilitat.

Tolerància al particionament de la xarxa La tolerància al particionament de la xarxa, ens indica la capacitat del sistema de suportar una fallida a la xarxa que provoqui la divisió del sistema. És a dir, que si en un moment donat es produeix una fallada a la xarxa i com a conseqüència el sistema es

divideix en dues parts incomunicades entre si, el sistema encara segueix sent operatiu.

HBase

HBase va ser la primera gran (coneguda) implementació del model orientat a columnes. Basada en un *paper* publicat per Google. HBase forma part del projecte Apache, i com a tal, està directament integrada amb altres eines del projecte. Així HBase en si únicament ofereix el model de dades, mentre que la replicació i l'escalabilitat les obté dels projectes HDFS (Hadoop) i Zookeeper respectivament. El fet d'estar composat per tres aplicacions diferents, fa que HBase sigui una mica complexe de configurar, però una vegada es sap com fer-ho, afegir o treure nodes d'un clúster, és una operació molt ràpida.



Listing 21: Esquema general d'HBase

Els tres pilars principals d'HBase son:

1. Garantir la consistència de les dades
2. Oferir una gran velocitat de resposta per a les lectures
3. Model de dades orientat a columnes

El primer objectiu es garanteix amb el disseny d'HBase, el fet de tenir un únic node responsable d'un "set"de dades, garanteix que les dades siguin consistents. Ja que qualsevol operació sobre una dada es portarà a terme per un únic node.

El segon objectiu va lligat a la integració d'HBase amb Hadoop. Hadoop és una altra eina d'Apache que es centra en l'anàlisi de grans quantitats de dades. El processat de dades en paral·lel permet a Hadoop analitzar grans quantitats de dades en temps molt petits. És per això que HBase busca oferir temps de lectura molt reduïts, per a oferir una integració total amb Hadoop.

El model de dades d'HBase és totalment orientat a columnes. Un model de dades orientat a columnes el que ens permet és tenir una família de columnes (taula al model relacional) fixada, però sense nombre de columnes fixat, el nombre de columnes és ilimitat. Això permet que qualsevol fila a una familia de columnes pugui tenir diferent nombre de columnes. Una fila pot tenir 2 columnes, i l'altre 2500, i això no implica que la que només en té 2 hagi de tenir 2498 *nulls*. Més endavant explorarem aquest model de dades en profunditat.

El set d'operacions disponibles sobre HBase és molt elemental, bàsicament es redueix a inserir una dada, i llegir una dada. També es disposa d'una operació de lectura de rangs que ofereix lectures més optimitzades, però aquestes son bàsicament les operacions que podem realitzar sobre HBase. No tenim operacions d'unió de taules, no tenim cap tipus d'ordenació ni d'agrupació.

HBase en resum, ofereix els 3 punts claus dels sistemes orientats a columnes, model de dades flexible, escalabilitat i replicació.

Cassandra

Cassandra és una eina que neix a Facebook, basada en una tecnologia creada per Amazon. El concepte de Cassandra és bàsicament el d'una taula de Hash distribuïda en que tots els nodes son iguals. Cassandra és un sistema orientat a columnes complert, és a dir, no depèn de cap altre sistema per funcionar.

Les característiques principals de Cassandra son:

1. Taula de Hash distribuïda

2. Flexibilitat en la configuració

3. Model de dades orientat a columnes

El primer punt, el que comporta és que tots els nodes del sistema siguin iguals. No hi ha cap node màster i cap node és l'encarregat d'unes dades en concret. Podem realitzar qualsevol operació sobre qualsevol node. Això no vol dir que tots els nodes tinguin totes les dades, sinó que son capaços de comunicar-se entre ells i obtenir el resultat desitjat.

Així com HBase és un sistema on clarament es busca obtenir un gran rendiment en les lectures, Cassandra és un sistema molt més flexible. Mitjançant la configuració podem decidir si desitgem potenciar el rendiment a les lectures o a les escriptures, modificant l'enfoc del teorema CAP que hem vist a aquest capítol.

Amb aquests dos primers punts, Cassandra ofereix un sistema molt flexible, però aquestes decisions de disseny comporten que Cassandra sigui un sistema que no garanteixi la consistència de les dades. Això no vol dir que no ho siguin, vol dir que està en mà del desenvolupador que ho siguin o no. Aquest és un punt que genera bastant controvèrsia entre els seguidors i els detractors de Cassandra.

En quant al model de dades, Cassandra ofereix un model totalment orientat a columnes, però més complert que el d'HBase. El model de dades de Cassandra introdueix un element que no trobem a HBase com son les super-columnes. Les super-columnes son un nivell superior a les columnes d'HBase. Una super-columna permet agrupar grups de columnes. Com veurem més endavant això permet obtenir models de dades més complexes que amb HBase.

Cassandra, al contrari que HBase ens ofereix molts tipus de consulta. Ens ofereix cerques de múltiples valors simultanis, ens ofereix ordenació dels valors obtinguts i moltes altres consultes que fan que el desenvolupador se senti més còmode si ve de fer servir SQL. En realitat la majoria d'aquestes crides, com a mínim a les seves primeres versions, son simples encapsulacions de les cerques bàsiques. És a dir, si fem una cerca de múltiples valors l'únic que fa és múltiples cerques dels valors individuals. Això fa que a vegades es tingui un concepte no del tot realista del funcionament de Cassandra. Com veurem més endavant és molt important conèixer una mica el funcionament intern i el model de dades per tal de treure'n el millor rendiment i el fet de disposar d'aquest tipus de consulta fa que no hi posem massa atenció i acabem sense treure el màxim rendiment de Cassandra.

2.3 Conclusions

Durant tot aquest capítol hem vist les principals característiques d'un model ja conegut, els sistemes relacionals, i d'un nou model, els sistemes NoSQL.

Una vegada vistes aquestes característiques, ha de quedar clar el perquè no podem comparar els sistemes relacionals amb els sistemes NoSQL, son dos sistemes amb enfocs diferents i objectius diferents. Els sistemes relacionals Ens ofereixen moltes garanties i molt control sobre les dades que contenen, les propietats ACID, les transaccions, etc. Treballen amb model de dades estàtic i ofereixen un llenguatge molt complert per a fer consultes. Però els sistemes relacionals no estan pensats per a treballar en entorns d'alta demanda.

Per altra banda, els sistemes NoSQL estan únicament pensats per a treballar en entorns d'alta demanda, i les seves característiques difereixen molt dels sistemes relacionals. No ens garanteixen les propietats ACID, no tenim transaccions i el llenguatge de consulta és limitat. El canvi de model de dades respecte al relacional provoca que hàgim de desenvolupar les aplicacions de forma diferent, pensant en les consultes més que en les dades, introduint redundància en cas de que sigui necessari. Però el seu objectiu que és donar suport a les aplicacions d'alta demanda, l'assoleixen perfectament.

Així tenim dos models de bases de dades, per a situacions diferents i amb característiques diferents. I com veurem als següents capítols dins dels sistemes NoSQL, cada solució té característiques diferents que la fan adient per a entorns específics.

Capítol 3

HBase

HBase, és dels primers sistemes que va néixer arrel de la publicació del paper de Google [1]. Tant és així, que es tracta d'una implementació gairebé literal del que s'exposa al paper, sense incorporar gaires novetats. Tota la implementació d'HBase està realitzada en Java.

HBase es presenta com a un sistema de codi lliure, distribuït, versionat i orientat a columnes basat en el paper de Google. Tal com diu la descripció d'aquest paper, l'objectiu és tenir un sistema d'emmagatzematge distribuït, per a tractar amb dades estructurades dissenyat per escalar-se amb volums de dades molt grans, arribant als PetaBytes. Tot això sobre un clúster de màquines estàndard (*commodity hardware*).

HBase forma part del projecte Hadoop d'apache, de fet és presentat com la base de dades de Hadoop. Com veurem més endavant, té una gran integració amb altres elements d'aquest projecte. Actualment HBase té un gran nombre de *commiters* al seu servidor d'svn i al seu canal per a desenvolupadors de l'IRC hi ha bastant activitat a qualsevol moment del dia. Per tant podem dir que és un projecte molt viu i en constant evolució.

Com que és un projecte en constant evolució, contínuament van apareixent noves característiques i això fa que probablement el dia que s'entregui aquest projecte, alguns dels aspectes comentats aquí, hagin evolucionat. Per tant, cal dir que tot el que es comenti al llarg d'aquest document en referència a HBase, serà basat en la versió 0.20.

3.1 Característiques principals

Abans de veure en detall totes les característiques d'HBase, fem-ne un repàs de les seves característiques principals i les diferències amb els SGBDR's.

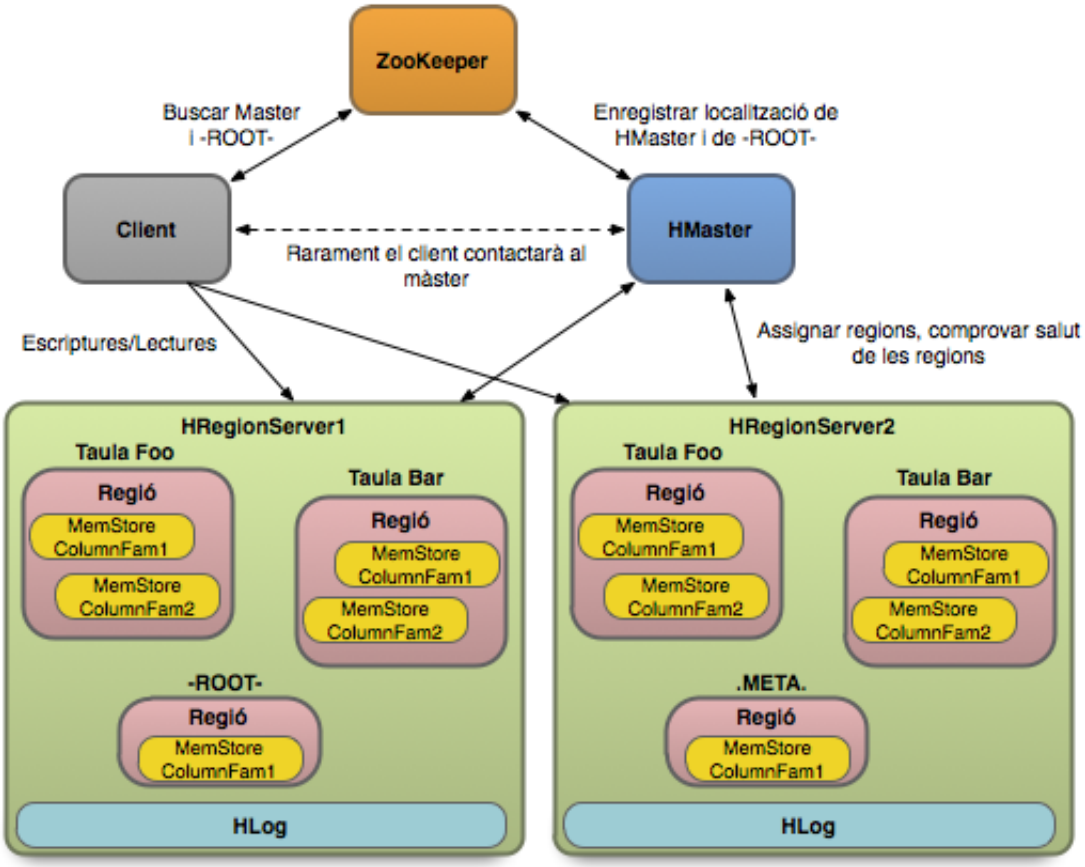
	SGBD Relacionals	HBase
Esquema de dades	Orientat a files	Orientat a columnes
Llenguatge de consulta	SQL	get/put/scan
Seguretat	Autenticació/Autorització	cap
Índexs	Columnes arbitràries	Només a la clau de la fila
Quantitat màxima de dades	TeraBytes	PetaBytes+
Throughput escriptura/lectura	Milers consultes/segon	Milions de consultes/segon
Garantíes ACID	si	no
Versionat de les dades	no	si
Teorema CAP	CP	CP
Ordenació	si	no

Listing 22: Caracterísitiques d'HBase i característiques dels SGBDR's

Durant els següents apartats farem un anàlisis a fons d'HBase i veurem en detall totes aquestes característiques.

3.2 Arquitectura

HBase està format per varis components com podem veure a la següent figura:



Listing 23: Estructura d'HBase

Aquesta figura [1] ens dona una visió global del que és HBase, però hi apareixen molts termes desconeguts per a qualsevol persona que no estigui familiaritzada amb HBase:

- Regió (*Region*): Una regió és un conjunt de files d'una taula.
- Servidor de Regions (*HRegionServer*): És el que gestiona les regions, serveix les dades de lectures i escriptures.
- Màster (*HMaster*): És el coordinador de les regions. També serveix per a realitzar tasques administratives.

[1] La majoria d'imatges d'aquest capítol son propietat de Cloudera, per comoditat del lector m'he permès fer-ne la traducció al Català

- META: És la taula que emmagatzema la informació de totes les regions i la seva localització.
- ROOT: És la taula que fem servir per a localitzar totes les taules META.
- Zookeeper: Té la informació global sobre el clúster.

Per a fer-ho el més entenedor possible, passem a analitzar les parts que formen HBase començant per les regions i acabant pel màster. Per últim introduirem el client i un component exterior a HBase anomenat Zookeeper.

3.2.1 Region

Les regions d'HBase son els elements finals de tota la cadena, son les dades en si. Una regió no és més que un conjunt de files ordenades d'una taula. Un conjunt de regions, ordenades adequadament, formen una taula. L'ordre de les files, com ja hem vist, ve donat per la clau de la fila i el particionador que fem servir.
Una regió te dos estats diferents:

- Memòria
- Disc

L'estat en memòria s'anomena *Memcache*. Memcache és un conjunt de files ordenades a una estructura de dades (MemStore de Java). Al tenir aquesta estructura de dades en memòria, l'accés és molt més ràpid que si ho tinguéssim a disc. Per tant les operacions d'escriptura es realitzen primer sobre memòria. Una vegada aquesta cache assoleix una mida límit, normalment de 64 *MegaBytes*, la informació passa al segon estat.

El segon estat és l'emmagatzemament al disc dur. Es fa un bolcat de tot el contingut de Memcache a un fitxer anomenat MapFile. Aquest fitxer es desa sobre un sistema de fitxers distribuïts anomenat HDFS (Hadoop File System), que és l'encarregat de fer la replicació de les dades.

La raó de tenir aquesta Memcache abans de passar a l'escriptura en disc, és simplement per raons d'eficiència. És molt més eficient fer escriptures a memòria i fer una única escriptura seqüencial (les files estan ordenades) a disc, que haver de fer una escriptura a disc per cada fila que volem escriure.

És molt important no associar una regió amb una taula a un SGBDR. Una regió és un conjunt de files d'una taula, però no conté totes les files d'una taula. Una taula pot estar dividida en múltiples regions, i cada regió com veurem pot estar associada a un servidor diferent. Podem veure'n un

exemple a la figura 23, on per exemple la taula foo, està dividida en dues regions.

3.2.2 HRegionServer

Cada regió està assignada a un (i només un) servidor de regions, per tant totes les peticions per a una mateixa regió sempre son dirigides al servidor encarregat d'aquella regió en aquell moment. Aquest és un dels detalls estructurals més importants d'HBase, ja que el fet de que una regió només pugui ser modificada per un servidor, és el que garanteix la consistència de les dades [2]. Com que un únic servidor farà totes les escriptures i les lectures, qualsevol lectura posterior a una escriptura tindrà la versió més recent de la dada.

El servidor de regions és l'encarregat de mantenir totes les regions que té assignades. Per tant, és el que fa les operacions d'escriptura i lectura, així com altres operacions que es detallen a continuació.

Lectura

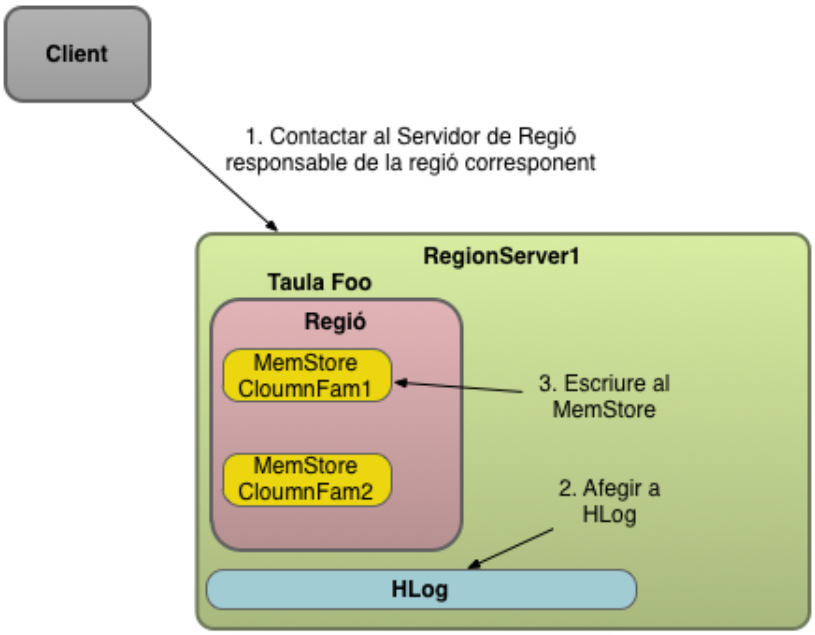
Quan el client demana una fila, la seva petició és re-dirigida al servidor de regió encarregat de la regió que conté la fila en qüestió. Aleshores, si la fila es troba a la Memcache, es serveix directament de memòria, sinó es busca a disc i es serveix.

Escriptura

El procés d'escriptura és senzill, indiquem la clau de la fila que volem escriure, es re-dirigeix la petició al servidor de regions que conté la regió a on es troba el rang de claus a on s'ha d'inserir la nova fila, i s'escriu la fila a la Memcache d'aquesta regió.

Si el procés fos tan senzill com hem explicat, HBase tindria un forat important, ¿què passaria si el servidor de regions caigues sense haver buidat el contingut de la Memcache a disc? Amb el sistema tal i com l'hem descrit fins ara les dades simplement es perdrien, i això és absolutament inadmissible. Així que existeix un element intermig anomenat HLog, que és l'equivalent al dietari dels SGBDR's. Cada escriptura es realitza primer a l'HLog i després a la Memcache.

[2] La C al teorema CAP



Listing 24: Procés d'escriptura a HBase

l'HLog a diferència de la Memcache és comú per a tot el servidor de regions, és a dir, l'HLog d'un servidor de regions conté les escriptures realitzades a totes les regions d'aquell servidor. Mentre que Memcache només conté les escriptures realitzades a una única regió.

El fet de tenir aquests HLogs ens serveix per a poder recuperar la informació en cas de caiguda d'un servidor de regió. És important aclarir, que el fet de que una regió estigui assignada només a un servidor de regions, no vol dir que només n'existeixi una còpia. Com hem vist abans, HDFS s'encarrega de fer les rèpliques de les regions així com dels diferents HLogs. El procés de recuperació en cas de caiguda d'un servidor de regions és el següent:

1. S'obté d'HDFS l'HLog del servidor que ha caigut.
2. Es separa l'HLog en diferents HLog's, un per a cada regió.
3. S'assigna un nou servidor per a la regió afectada i es reprodueix tot el contingut de l'Hlog sobre la Memcache d'aquesta regió.
4. El servidor de regions actualitza la taula META i n'informa al màster.
5. El nou servidor de regió ja està preparat per a començar a servir la regió amb tota la informació actualitzada.

Eliminar files

Eliminar no és considerat una operació, ja que per a eliminar el que es fa és simplement afegir un marcador a una fila que diu que s'ha d'eliminar. Per tant, una operació d'eliminació és de fet una escriptura. La eliminació en si es realitza al fer les compactacions.

Compactacions (*Compactation*)

Com hem vist, a mida que es vagin fent escriptures, s'aniran creant MapFiles de 64MegaBytes. Però 64 MegaBytes és una mida relativament petita, i no és del tot eficient, ja que per a buscar una fila potser hem de buscar a dins de varis fitxers, o si fem un scan, també haurem d'usar varis fitxers. Cal remarcar que MapFile no és sinònim de regió, una regió pot estar formada per varis MapFiles, així com una taula pot estar formada per vàries regions.

Hi ha dues operacions que serveixen per a unir (compactar) aquests fitxers.

- *Minor Compaction*: S'executa quan s'arriba a un cert nombre, configurable, de fitxers. Aquesta operació agafa fitxers petits i els ajunta en un de més gran. Aquesta operació pot resultar en varis fitxers de diferents mides.
- *Major Compaction*: Aquesta operació és igual que l'anterior, però aquesta treballa amb els fitxers grans creats amb les *minor compactions*. Com que treballar amb fitxers grans és més costós, aquesta operació només s'executa una vegada per dia i d'aquesta operació en resulta un únic fitxer.

Separacions (*Split*)

De la compactació de fitxers en poden resultar fitxers massa grans. Hi ha un límit, configurable, per a la mida dels fitxers, i una vegada es sobrepassa aquest límit el que es fa és dividir el fitxer en dos, o el que és el mateix, separar una regió en dues regions.

1. Una regió arriba a la mida límit (256 MegaBytes per defecte).
2. El servidor de regions divideix la regió en 2 regions de mida igual (aproximadament).
3. El servidor de regions actualitza la taula META i n'informa al màster.
4. El màster assigna la nova regió a un servidor de regions.

3.2.3 HMaster

El màster d'HBase és l'encarregat de gestionar les regions del clúster, però no és un màster en el sentit tradicional de la paraula. A diferència del màster d'un SGBDR, les operacions sobre les regions no han de passar per ell. Per tant el màster no serà mai el coll d'ampolla del sistema. És per això que un nom més convenient per al màster seria el d'organitzador. Això no treu que el màster hagi estat el punt únic de fallida d'HBase durant molt de temps, però això ja ha estat corregit amb la versió 0.20

El màster és l'encarregat de gestionar les regions, això vol dir afegir-ne, eliminar-ne, modificar-ne l'esquema (*column families*), etc. El màster disposa de dues regions del sistema per a fer aquestes operacions, ROOT i META.

ROOT i META son dues regions especials del sistema que serveixen bàsicament per a localitzar les regions al sistema.Qualsevol operació sobre regions, tal com crear, eliminar, dividir, etc. Implica realitzar les modificacions necessàries a ROOT i META, i aquesta és la feina del màster

El màster també és l'encarregat de monitoritzar els servidors de regions, en cas de fallida d'un d'ells, és l'encarregat de re-assignar les regions contingudes a aquest servidor a d'altres servidors de regions. Aquesta operació és la que s'ha explicat al capítol anterior.

META

La taula de META és la que té la informació bàsica de totes les regions del sistema. En sap la localització, la primera i última fila, si la taula està activa o inactiva, etc. El fet d'afegir o eliminar regions, implica treure o afegir una taula a META. META és una regió com qualsevol altre del sistema, per tant pot crèixer tant com calgui i automàticament s'anirà dividint en altres regions.

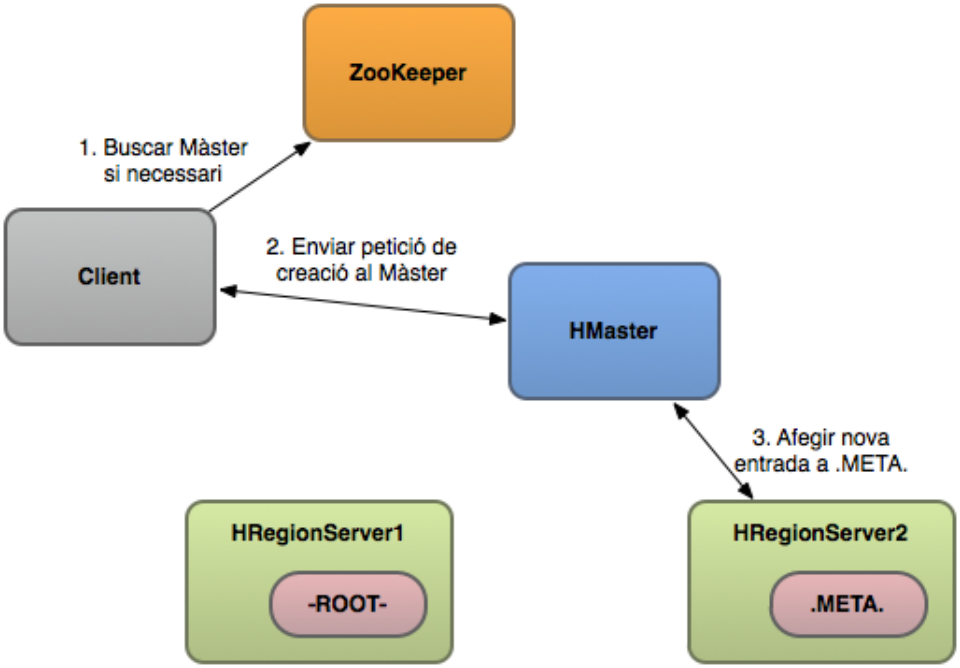
ROOT

La taula de ROOT només ocupa una regió al sistema, no pot créixer més. ROOT serveix per a localitzar totes les regions de META.

Una fila de ROOT ocupa aproximadament 1KB, si tenim la mida de regió configurada a 256MB (per defecte), això vol dir que ROOT pot mapejar $2.6x10^5$ regions de META, que equival a $6.9x10^{10}$ regions d'usuari, que és el mateix que $1.8x10^{19}(2^{64})$ bytes de dades.

3.2.4 Client

El client és el que realitza operacions sobre HBase, però per a poder realitzar una operació sobre una regió, primer ha de localitzar la regió. Per a localitzar la regió com hem vist, ens cal la taula de META, però per a trobar META ens cal ROOT, i la pregunta és ¿Com trobem ROOT? La resposta és amb Zookeeper. Zookeeper és un dels components del projecte Hadoop d'Apache. És l'encarregat de gestionar el clúster, i és ell el que coneix la localització de la regió ROOT.



Listing 25: Procés de creació d'una taula a HBase

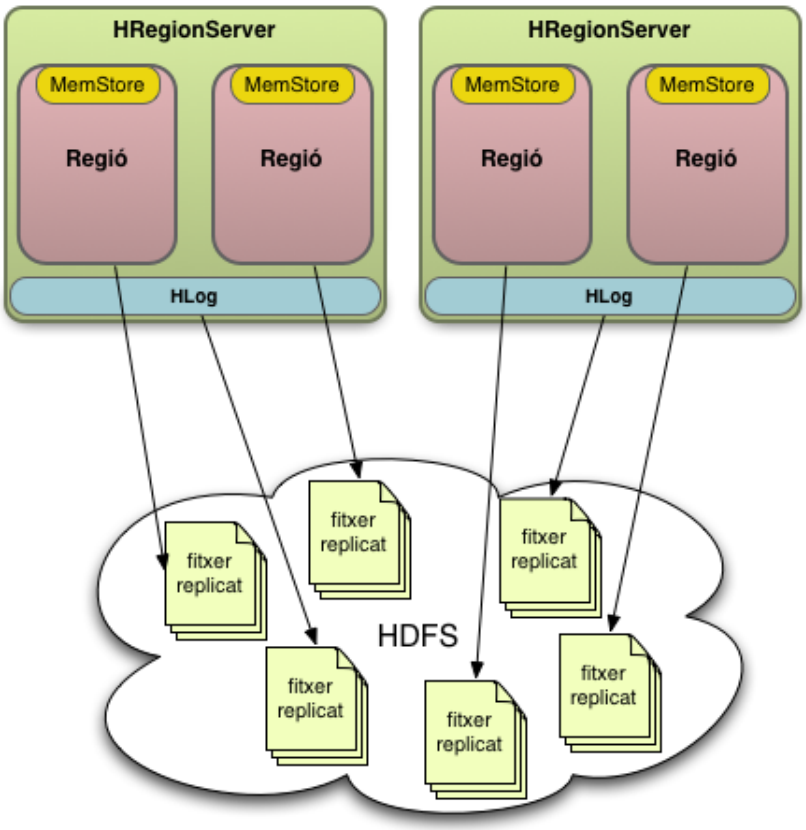
El client d'HBase té únicament 3 operacions disponibles:

- Get: Obté una fila
- Put: Desa una fila
- Scan: Especificant una fila d'inici i fi, obté un rang de files

Com és d'imaginar, l'operació més útil d'HBase és l'scan, ja que ens permet usar filtres i opcions que la fan una eina molt potent.

3.2.5 Replicació de les dades

Com hem vist anteriorment, HBase només s'encarrega de proporcionar el model de dades, no s'encarrega de la replicació de les dades. La replicació de les dades la fa HDFS, el sistema de fitxers del projecte Hadoop d'Apache.



Listing 26: Integració d'HBase amb HDFS

Com es mostra a la figura 26, les regions físicament les maneja HDFS i és qui s'encarrega de fer-ne múltiples [3] còpies i distribuir-les pels nodes. Hi ha un inconvenient de que HBase no sigui el gestor dels fitxers, i és a l'arrencada del sistema. Si aturem el clúster i el tornem a arrencar, al moment d'anomenar els servidors de regió, HBase ho fa aleatòriament. Per tant el més provable és que el node encarregat de servir una regió no disposi físicament de les dades. Això comporta que quan demanem una dada al servidor de

[3] 3 per defecte

regió, aquest l'hagi de demanar a un altre node, amb la pèrdua de rendiment que això comporta. El que fa HBase, és forçar a HDFS a desar una còpia local al servidor de regió, al moment en que aquest faci una compactació.

En definitiva, al cap d'una estona[4] de funcionar, quan s'hagin fet particions de totes les regions, HBase garanteix que cada servidor de regió, conté una de les rèpliques d'HDFS. Per tant, els clústers d'HBase funcionen molt millor quan porten molt temps sense ser apagats.

3.3 Model de dades

Com hem vist a la descripció, HBase té una estructura de dades a priori similar als SGBDR's, taules, files i columnes. Però el model de dades no s'assembla al dels SGBDR's. El canvi principal, com és d'imaginar ve quan parlem de columnes, per alguna cosa s'anomenen sistemes basats en columnes.

Una fila d'HBase està composta per 3 elements:

- La clau de la fila
- Un conjunt de columnes
- Una marca de temps (*timestamp*)

3.3.1 Claus

La part més important del model de dades d'HBase (i la resta de sistemes NoSQL), és la clau d'una fila. La clau, és l'equivalent a la *primary key* de la taula a un SGBDR, i és el que ens permet trobar una fila a una taula.

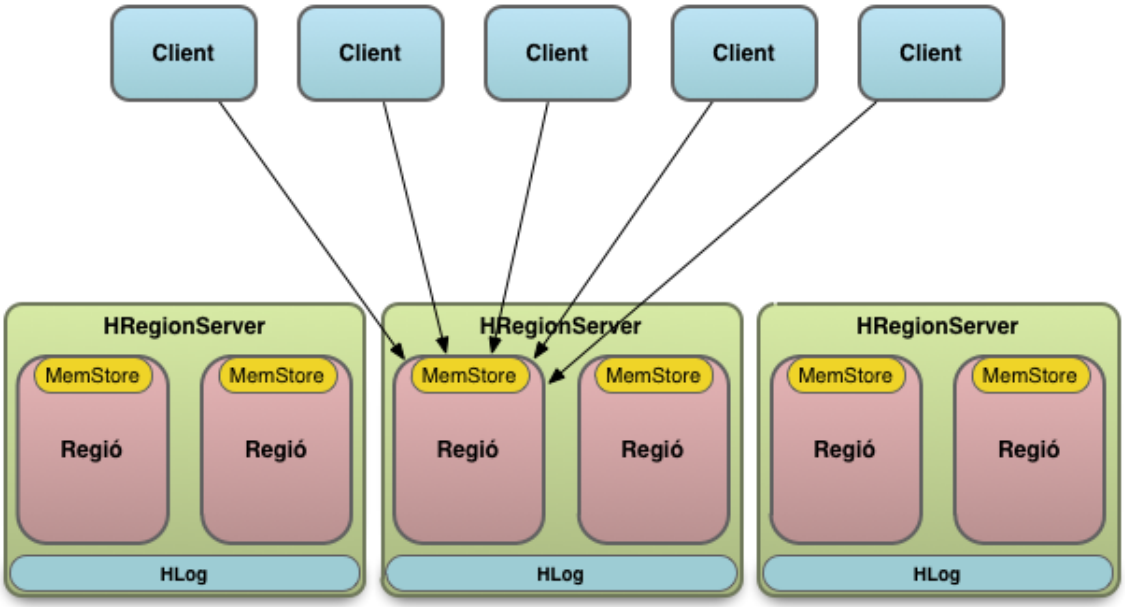
Com hem vist al capítol anterior, la clau d'una fila juga un paper molt important ja que és la que determina a quina regió s'inserirà la fila i per tant dona una localitat a les dades. Aquesta localitat de les dades, com ja hem vist, ve donada pels particionadors. HBase a diferència de Cassandra, només disposa d'un particionador i no té cap opció per a fer-ne servir un altre. El particionador que fa servir HBase és el particionador per ordre lexicogràfic.

Particionador

El fet de que HBase sempre ordeni les dades lexicogràficament és un repte a l'hora de dissenyar els models de dades, ja que com hem vist el correcte repartiment de les dades és clau per al bon funcionament del sistema.

[4] Potser dies

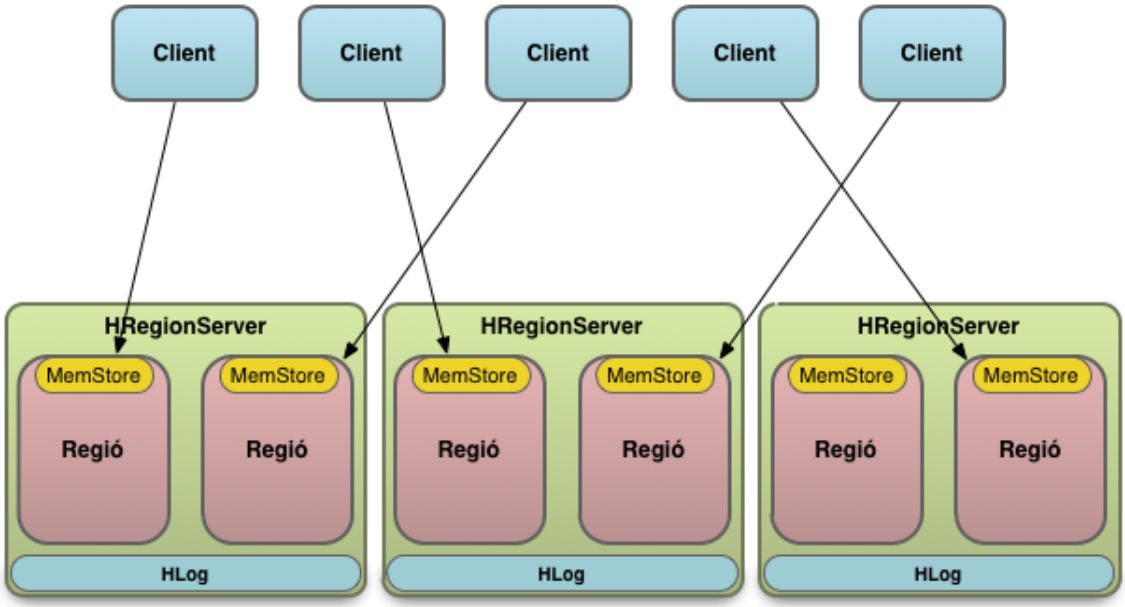
Agafem un exemple real per a mostrar com adaptar les claus al particionador d'HBase és eseencial per a obtenir un bon funcionament. Una empresa tenia un sistema de notificació d'errors al seu programa, cada vegada que un usuari premia al botó per a notificar als desenvolupadors, s'enviava el problema als servidors de la companyia. Al arribar a la companyia, es desaven les incidències sobre HBase, usant com a clau de la fila el *timestamp* del moment en que s'havia generat l'error. Els *timestamps* son nombres molt llargs, i per tant les seves primeres posicions varien molt poc. Així l'ordenació lexicogràfica feia que tots els errors anessin a parar a un únic servidor, mentre que els altres no tenien cap tipus de càrrega.



Listing 27: Particionador amb mala elecció de clau

Aquesta elecció de clau, feia que tinguessin una infraestructura molt poc optimitzada, ja que un únic servidor era l'encarregat de gestionar tota la informació.

El que van fer, va ser posar un bit aleatori al davant del *timestamp*, així en comptes d'ordenar-se pel *timestamp*, s'ordenaven segons aquest bit i aconseguien una distribució totalment aleatòria de les dades.



Listing 28: Particionador amb bona elecció de clau

L'únic inconvenient d'aquesta segona versió és que si es volen obtenir totes les incidències per una única data, aquestes estaran repartides per tots els nodes, per tant serà una consulta bastant ineficient. Però l'empresa en qüestió va veure que rebien moltes incidències al llarg d'un dia, però ells només feien la consulta de totes les consultes rebudes a una certa data de tant en tant. Per tant els afavoria molt més una solució que beneficiés les escriptures tot i que penalitzés les lectures.

3.3.2 Columnes

Les columnes a HBase no son iguals que al model relacional. Les columnes a HBase tenen la forma <família>:<etiqueta>, on les dues parts son un *array* qualsevol de bits. La part <família> és el que s'anomena *column family* i ha de ser declarada explícitament al crear la taula. Podem en canvi crear etiquetes (columnes) en qualsevol moment amb una simple instrucció de write.

Al model relacional, una columna, té un valor per fila (i viceversa) sempre, inclús si no li volem assignen un valor, se li posarà *null*, que és un valor en si. Amb HBase això no és així.

Veiem un exemple, en un model relacional podríem tenir la següent taula

Usuari:

Clau	nom	email	telefon
1	Joan	joan@est.fib.upc.edi	921234567

Listing 29: Taula al model relacional

Aquesta taula té 4 columnes i un valor per a cada columna, però si afegim un nou registre amb dades incompletes de forma que tinguem la següent informació:

Clau	nom	email	telefon
1	Joan	joan@est.fib.upc.edi	921234567
2	Maria	maria@est.fib.upc.edu	null

Listing 30: Taula amb null al model relacional

Per aquesta nova fila el que farà l'SGBDR és emmagatzemar un *null* a la columna telèfon. Això té dues conseqüències:

- Ocuparà espai
- Pot crear problemes amb les cerques. A l'hora de crear consultes s'ha de ser conscient de si hi haurà *nulls* i gestionar-los correctament

Veiem ara com emmagatzemen les dades els sistemes de bases de dades orientats a columnes. Com el seu nom indica, les columnes juguen un rol molt important, el que fan és emmagatzemar els valors de cada columna per separat, així, si una fila no té valor per a una columna, simplement no s'emmagatzemarà res. No s'ocuparà l'espai extra que implica el valor *null*. Així, en un sistema NoSQL, la taula del segon exemple, quedaria de la següent manera comptant que la clau sigui l'id de l'usuari:

Clau	Timestamp	Columna "nom:"
1	t1	Joan
2	t1	Maria

Clau	Timestamp	Columna "email:"
1	t1	joan@est.fib.upc.edu
2	t1	maria@est.fib.upc.edu

Clau	Timestamp	Columna "telefon:"
1	t1	931234567

Listing 31: Taula amb null al model relacional

Així doncs quan parlem de que les columnes es poden crear al fer un simple write, i veient que el fet de tenir columnes només afecta en el cas de que posem un valor a una columna, tenim que podem crear tantes columnes com ens sigui necessari en qualsevol moment. Així per exemple si a aquesta taula d'usuaris creem una *column family*, que es digui informació, hi podrem afegir les columnes que ens convingui per a cada usuari.

Podríem tenir per a un usuari la columna "informacio:edat", "informacio:estatCivil", "nformacio:feina", "informacio:estudis", etc. Mentre que per a un altre usuari no tinguéssim cap d'aquestes columnes però si ïnformació:ciutat".

Aquesta flexibilitat amb les columnes, és el que es diu *flexible schema*, i és la major diferència respecte als SGBD's relacionals.

3.3.3 *Timestamps*

Els *timestamps* d'HBase existeixen perquè HBase és un sistema versionat, és a dir, que guarda diferents versions de les dades que conté. Així doncs si modifiquem una columna, HBase desarà el valor actual però també el valor anterior. Els valors es desen en ordre descendent segons el seu *timestamp*, així que per defecte sempre obtindrem el valor més recent. Això ens porta al fet de que tot i que el *timestamp* sigui necessari per a obtenir una dada, no cal que l'indiquem al fer el *get* a no ser que vulguem una versió concreta de la dada. Quan fem un *insert* podem proporcionar un *timestamp* si volem,

però si no ho fem es posa el *timestamp* actual del sistema a l'hora de fer l'escriptura.

Per defecte HBase guarda 3 versions per valor, però és un paràmetre que podem especificar per a cada taula i per tant podem posar el valor que ens convingui.

Veiem un exemple de com funcionen els *timestamps*. Si tenim la taula Usuaris ja descrita abans, podríem tenir:

Clau	Timestamp	Columna "telefon:"
1	t1	931234567
2	t1	939876543
2	t2	939876542
1	t3	931234568

Listing 32: Timestamps a una taula d'HBase

A la figura 32 es veuen les diferents versions que s'han desat per a la columna telèfon en diferents instants de temps, sent les versions més recents la del t2 per a l'usuari 2, i t3 per a l'usuari 1.

3.4 Conclusions

HBase va ser la primera eina NoSQL en aparèixer, i per tant va haver de fer front a un gran nombre de problemes que no havien existit anteriorment. Però HBase ha esdevingut una eina molt potent, molt coneguda i àmpliament usada al mercat actual.

HBase destaca amb una sèrie de punts forts sobre el que fins ara eren les eines principals al mercat, els SGBDR's.

Un dels punts forts d'HBase és el seu ratio d'escriptura que és de milers de files per segon per node. Això ens dona capacitat per desar grans quantitats d'informació en tems real. Una de les aplicacions més clara d'aquesta capacitat torna a ser la d'emmagatzemar logs d'altres sistemes.

Una de les eines més útils d'HBase és l'operació d'scan. És una eina molt potent, que juntament amb els filtres i un bon esquema de dades, permet (per la localitat de les dades), fan que es puguin realitzar consultes relativament complexes en temps molt baixos.

Potser el punt més a favor d'HBase tot i que no és una característica d'HBase en si, és la seva integració amb altres eienes de la fundació Apache. En especial amb Hadoop. Hadoop és actualment la gran líder al mercat en l'anàlisi de dades, i la integració total d'HBase amb Hadoop (comparteixen sistema de fitxers), fa que HBase passi a ser la primera opció al moment en que una empresa que ja usa Hadoop decideix usar una eina NoSQL. A part d'aquesta integració, cada vegada existeixen més eines per treballar sobre l'ecosistema Hadoop, com Pig o Hive, que son eines que busquen oferir llenguatges de consulta similars al SQL a sobre de Hadoop. L'èxit d'HBase per tant, va molt lligat a l'èxit de Hadoop i les seves eines associades.

Amb l'arribada de múltiples eines NoSQL, HBase va perdre bastant protagonisme, en front d'eines com Cassandra que ràpidament van crèixer en popularitat. Però l'anunci a l'ultim trimestre de 2010 que va fer Facebook dient que estava treballant en dues noves eines sobre HBase, quan Cassandra és una eina construïda per Facebook, va fer que HBase tornés a guanyar protagonisme i ara estigui sent de nou l'eina NoSQL amb més projecció.

Fins ara l'ús més indicat per a usar HBase ha estat per emmagatzemar logs per poder-ne fer l'anàlisi després amb Hadoop. I tot i que HBase ofereix molt més que això, sembla que és una mica al sector on s'ha enfocat. Tot i que amb novetats incloses a les noves versions, així com amb l'anunci de Facebook de que farà aquesta nova aplicació de missatgeria sobre HBase, sembla que altres empreses s'animaran a donar a HBase usos més enfocats al desenvolupament d'aplicacions.

Capítol 4

Cassandra

4.1 Característiques principals

Cassandra és una base de dades que va ser desenvolupada per Facebook. Concretament la van desenvolupar per a emmagatzemar el sistema de missatgeria entre usuaris. Facebook, una vegada va tenir desenvolupat Cassandra, la va alliberar com a projecte de codi lliure al 2008 a la web de Google Code. Al 2009 va ser acceptat com a projecte en incubació a la fundació Apache, i al Febrer del 2010 va passar a ser un projecte propi (el màxim nivell) a dins d'Apache.

Així com HBase és una adaptació gairebé literal de BigTable de Google, Cassandra és una adaptació del sistema Dynamo d'Amazon. Com veurem tot i que Cassandra i HBase son dos sistemes NoSQL que comparteixen uns mateixos objectius i característiques, internament son absolutament diferents.

Així com Hbase és un sistema rígid, amb unes característiques molt definides, Cassandra és un sistema molt flexible, que premet a l'usuari prendre moltes decissions respecte el seu funcionament. Així per exemple Cassandra per definició és un sistema que potencia les escriptures, però si l'usuari ho decideix, pot tenir un sistema sobre Cassandra que busqui el rendiment sobre més lectures més que sobre les lectures, simplement canviant alguns paràmetres de configuració.

Cassandra com veurem, disposa de diferents elements i eines que fan que sigui un sistema NoSQL complert. És a dir, cassandra no depen de cap altra eina. Cassandra incorpora el model de dades, la replicació, el particionat de les dades, etc.

Cassandra és considerat un sistema OLTP, és a dir, un sistema enfocat a realitzar aplicacions més que anàlisi de dades. Mentre que com hem vist HBase és un sistema que bàsicament es fa servir per l'anàlisi de grans quan-

titats de dades, Cassandra està pensat per a ser la base de dades de les aplicacions. On es desi tota la informació del model de les aplicacions.

4.2 CAP

El teorema de CAP com hem vist a capítols anteriors, aplica a tots els sistemes distribuïts i ens diu que de les 3 característiques (consistència, disponibilitat i tolerància al particionament de la xarxa), només en podem oferir 2 per tal de tenir una latència acceptable. Aquest teorema qualsevol de les eines NoSQL, l'apliquen a l'hora de desenvolupar l'eina i l'usuari no ha ni de tenir consciència de que existeix. Per exemple HBase tria CP, i mai s'exposa a cap document, això és perquè simplement és una decisió de disseny com tantes altres es prenen al desenvolupar qualsevol eina. Però a la que algú coneix Cassandra, coneix aquest teorema, i això és per la peculiar aplicació del teorema de CAP que fa Cassandra.

El teorema de CAP, si es mira bé, el que diu és que un algorisme distribuït només pot oferir dues de les 3 característiques, no un sistema en general sinó els algorismes que aquests fa servir.

Cassandra com a base ofereix AP, deixant de banda la consistència. Això per a qualsevol desenvolupador és segurament inadmissible, més encara quan estem parlant de bases de dades. Si una base de dades no és Consistent, podem estar escrivint dades per una banda i llegint-les després, i que el sistema ens retorni dades totalment fora de lloc (antiquades). De fet no és que Cassandra no sigui consistent sinó que és eventualment consistent, el que vol dir és que no garanteix la consistència, però que l'acabarà assolint. Això segurament per a molts desenvolupadors segueix sent inadmissible.

Si passem per alt la mala impressió causada per la falta de consistència, el que veiem és que Cassandra el que fa és oferir a l'usuari la possibilitat de triar el grau de consistència que vol tenir. Com a base es garanteixen la disponibilitat i la tolerància al particionament de la xarxa, però el grau de consistència pot ser definit per l'usuari, i no en termes generals, sinó en cada operació que realitza.

Per a oferir aquesta flexibilitat Cassandra ofereix 3 nivells de consistència:

- ONE: Escriptura/Lectura a un node
- ALL: Escriptura/Lectura a tots els nodes
- QUORUM: Escriptura/Lectura a (N/2)+1 Nodes, sent N el nombre total de Nodes

Per tant la consistència de les dades depèn de la combinació de modes de consistència que el desenvolupador trii per a les seves operacions. Anem a veure diferents combinacions per a garantir la consistència, per a fer-ho usarem la següent nomenclatura:

- R = Nombre de nodes on es realitza la lectura
- W = Nombre de nodes on es realitza l'escriptura
- N = Nombre de rèpliques de les dades
- Q = Quorum, o el que és el mateix (Q = N / 2) + 1

Si per exemple posem R = 1, volem dir que el sistema escriu a un node i dona l'escriptura per finalitzada, per tant, garanteix l'escriptura a un i només un node. Si R = Q, el sistema esperarà a escriure a Q Nodes abans de donar l'escriptura per finalitzada. Cal destacar que N no és el nombre de nodes del clúster, sinó un nombre definit per l'usuari. N defineix quantes còpies es vol tenir de les dades, és el factor de replicació.

Obtindrem consistència quan tinguem $R + W > N$. Per a veure que això és cert mirem per exemple si agafem N= 4 , R = 3 i W = 2. Veiem que si escrivim a dos nodes, però llegim a 3, com a mínim estarem llegint a un dels nodes on s'ha escrit, per tant el sistema veurà quina és la dada més recent i ens la retornarà.

3 diferents combinacions per a obtenir consistència que son habituals son:

- W=1, R=N
- W=N, R=1
- W=Q, R=Q

Amb aquestes diferents combinacions, el que fa Cassandra es donar a l'usuari el control de quines operacions vol potenciar. Per exemple, si ens interessa tenir una latència d'escriptura molt alta, el millor serà triar W=1, tot i que després la latència d'escriptura es veurà perjudicada.

Cal notar que no és obligatori fer servir aquestes operacions, podem tenir R=1 i W=1 a les nostres operacions, el que passa és que aleshores Cassandra no garanteix la consistència de les dades. Pot ser que obtinguem els resultats correctes al fer lectures, però pot ser que no.

Així doncs queda vist com Cassandra fa un ús peculiar del teorema de CAP, i permet a l'usuari regular la consistència de les seves dades. És un dels punts més discutits i molta gent qüestiona la tria de CP deixant de banda A, però el fet d'oferir aquesta flexibilitat deixa a les mans del desenvolupador el nivell de consistència desitjat, i moltes vegades permet obtenir grans latències en les operacions més conflictives.

4.3 Arquitectura

En quant a la implementació, Cassandra difereix molt d'HBase i BigTable. Mentre que les dues últimes fan servir un sistema de fitxers distribuït (GFS o HDFS), i una estructura amb nodes màster, servidors de regió, etc. Cassandra canvia el model del sistema distribuït, passant a usar una estructura basada en Dynamo i Gossip.

Dynamo és en si una base de dades distribuïda dissenyada per Amazon i de la qual en van publicar un paper explicant-ne la implementació [2]. Dynamo és una taula de Hash distribuïda. Gairebé tota l'estructura de Cassandra és idèntica a la de Dynamo, en aquest document es farà un repàs de totes les seves característiques, però per profunditzar en detalls més tècnics es recomana la lectura del paper sobre Dynamo.

Gossip per altra banda és un sistema de comunicació basat en p2p, que permet realitzar les tasques de comunicació entre nodes.

El punt central de Cassandra és el canvi d'orientació respecte d'altres bases de dades, tant SGBDR's com sistemes NoSQL. El que proposa Cassandra és tenir un sistema altament disponible, distribuït i amb baixa latència. La baixa latència és bàsica per a les empreses, sobretot per a complir els contractes SLA [1] que els obliga a complir certes condicions de latència en els seus sistemes. I com és sabut, tractar amb sistemes on la possibilitat de fallida és alta (tant de hardware com de xarxa), i a sobre mantenint una alta consistència de les dades, és contrari a obtenir altes latències.

Així que el canvi que fa Cassandra (Dynamo) és plantejar un sistema amb varis aspectes centrals:

- Sistema eventualment consistent, és a dir, no garanteix la consistència de les dades, però a canvi busca obtenir una alta disponibilitat de les dades.

- Escalabilitat incremental: S'han de poder afegir nodes sense haver de fer modificacions al sistema.

- Simetria: Tots els nodes han de ser idèntics. Cada node ha de tenir exactament les mateixes responsabilitats.

- Descentralització: No hi ha d'haver cap node central. El disseny ha de buscar l'ús de tècniques p2p i evitar qualsevol tipus de control centralitzat.

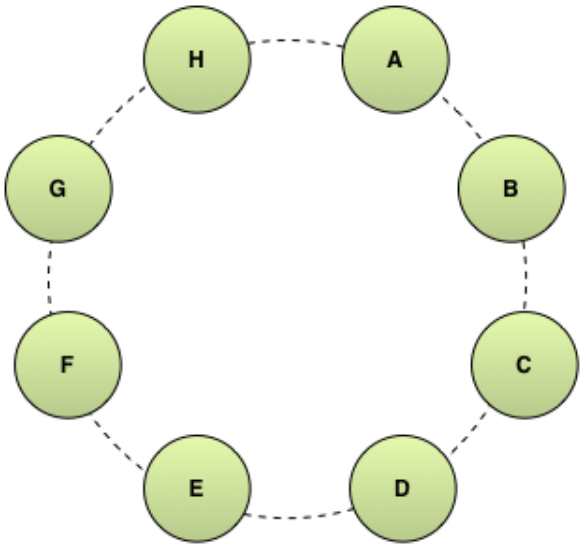
[1] Service Level Agreement

- Heterogeneïtat: La carrega de treball ha de ser proporcional a la capacitat dels servidors. Així es poden afegir nous servidors amb característiques diferents als ja existents.

Amb aquest canvi d'orientació i el fet de ser una taula de Hash distribuïda, ens dona un sistema amb una arquitectura totalment diferent a a HBase.

4.3.1 Estructura

Cassandra és en si una taula de Hash distribuïda circular. Això el que vol dir és que cada node té un node predecessor i un successor, formant una estructura de cercl com es pot veure a la figura 33.



Listing 33: Estructura en cercle

Una taula de Hash és una estructura de dades que emmagatzema parelles clau-valor. Per a obtenir la clau, es disposa d'una funció que a partir d'un valor ens genera una clau. Un mateix valor sempre generarà la mateixa clau. Això ens dona dues operacions, desar clau i obtenir clau. Aquestes dues operacions tenen cost O(1).

Com veurem més endavant, Cassandra ens ofereix més que desar un valor, ens ofereix desar una estructura de dades similar a HBase, però això ho veurem quan estudiem el model de dades de Cassandra.

Hem parlat de les taules de Hash, però Cassandra no és únicament una taula de Hash, és una taula de Hash distribuïda. Això implica que està

formada per vàries parts (nodes). El comportament és exactament el mateix que una taula de Hash normal, però en comptes de desar els valors a una única estructura de dades, cada node de Cassandra conté una part d'aquesta estructura de dades. Cada node de la figura 33 conté una part de les dades de la taula de Hash que forma Cassandra.

Com a qualsevol sistema distribuït, cal comunicació entre els nodes. Cassandra fa servir un sistema basat en comunicació directa entre nodes (p2p) anomenat Gossip. Durant tot el capítol, qualsevol explicació que impliqui comunicació entre nodes, enviament de missatges, enviament de dades entre nodes, etc. Gossip serà l'encarregat de portar a terme aquesta comunicació.

4.3.2 Nodes

La característica principal dels nodes és la simetria. Tots els nodes de Cassandra son idèntics, tots tenen exactament les mateixes responsabilitats. Aquesta característica el que busca és evitar tenir un únic punt de fallida i evitar la configuració o el manteniment.

Un node té una estructura interna similar a la dels servidors de regió d'HBase. Té a part de les estructures de dades en disc, una caché on desa les dades en primera instància abans de fer-les persistents al disc. El que a HBase s'anomena MemCache a Cassandra s'anomena MemTable. Aquesta MemCache ens permet augmentar la velocitat d'escriptura de les dades, així com millorar la velocitat de lectura de les dades recents.

Com tots els sistemes NoSQL, la localitat de les dades juga un paper molt important a Cassandra. Com veurem Cassandra disposa de varis particionadors. Cada node de Cassandra té assignat un rang de dades, i n'és conscient. Així com a HBase es separava la regió i el servidor de regions, a Cassandra cada node sap el rang que té assignat, i disposa d'una còpia en local d'aquestes dades.

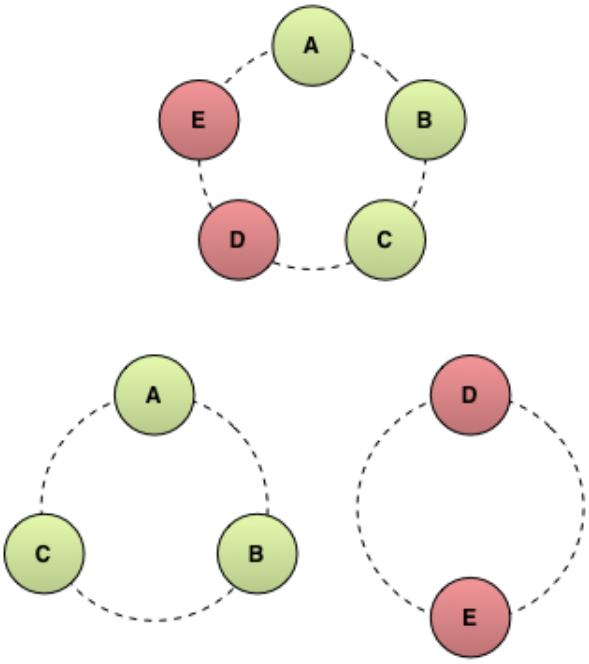
Per temes d'optimització a l'hora de realitzar operacions, cada node coneix els rangs de dades assignats a la resta de nodes, així si un node rep una petició per a una dada que no li correspon sap a quin node adreçar la consulta sense haver de consultar-ho. Això és molt pràctic ja que com hem dit, els nodes de Cassandra son els únics elements que té Cassandra. No hi ha cap tipus de mecanisme central, els clients es connecten directament als nodes. Per tant el més provable és que un client faci consultes al node al que està connectat, però que el node no sigui el responsable d'aquelles dades.

El que ens ofereix el disseny de nodes Cassandra, és un sistema amb escalabilitat lineal. Si tenim un clúster amb un únic node, i n'afegim un altre d'igual, passarem a tenir un clúster amb el doble de capacitat en tots els

sentits. Tindrem el doble d'espai d'emmagatzematge [2], el doble de capacitat de lectura, i el doble de capacitat d'escriptura.

Datacenters

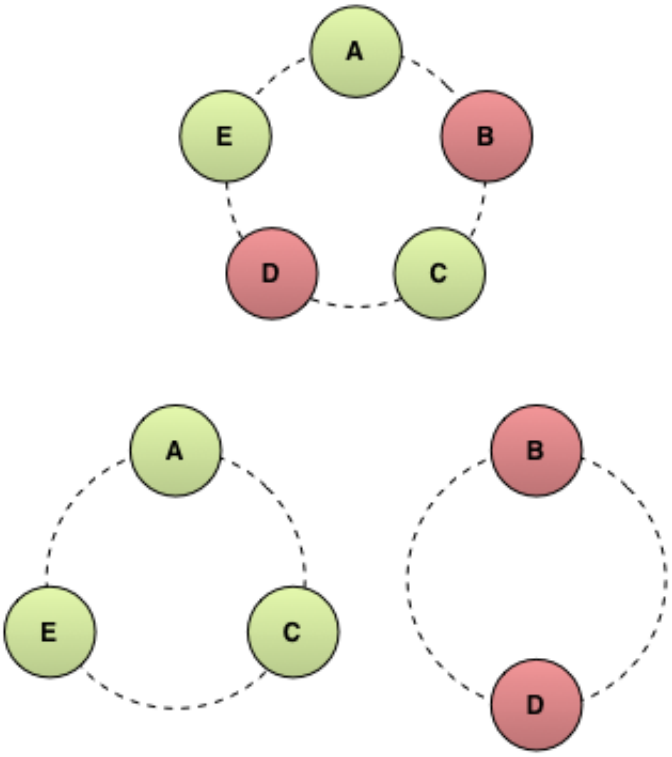
Com qualsevol sistema distribuït, Cassandra a part d'estar formada per varis nodes, pot estar distribuïda per varis *datacenters* que a l'hora estan formats per varis nodes.



Listing 34: Separació en datacenters

Si mirem la figura 34, disposem d'un sistema format per 5 nodes: A, B, C, D i E. A la vegada aquest sistema està distribuït a dos *datacenters* diferents D1 i D2. De tal manera que D1 conté els nodes A, B i C. Mentres que el D2 conté D i E. Aquesta és una topologia molt poc freqüent, ja que l'assignació segueix un ordre, el més habitual serà que la repartició de nodes en *datacenters* sigui menys ordenada.

[2] això no és cert degut a la replicació



Listing 35: Separació en datacenters

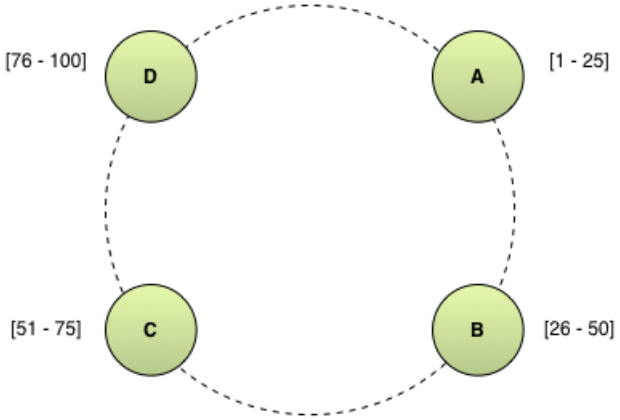
Com veiem a la figura 35, aquesta repartició en *datacenters*, no conserva l'ordre del del cercle dels nodes de Cassandra. El node B i D, es troben a un *datacenter* que no conté ni els seus antecessors ni els seus predecessors.

Aquesta repartició de nodes en *datacenters* la pot definir el desenvolupador mitjançant la configuració de Cassandra. I el més normal serà tenir una separació coherent amb el disseny. Com veurem més tard, Cassandra ens ofereix eines per a treballar amb aquesta separació en *datacenters*, com per exemple a l'hora de definir la replicació de les dades.

4.3.3 Particionament

Com a qualsevol sistema distribuït, és important poder trobar la informació en els menors passos possibles, i encara més si parlem d'una base de dades, per tant és molt important saber com es desa la informació i saber-la recuperar. Per això un node és responsable d'un rang de claus per a mantenir un ordre. Però aquest ordre no consisteix únicament en tenir un rang de claus, sinó en que aquest rang estigui ordenat. Per tant el rang de claus assignat ha de ser

un rang ordenat. I no només això, sinó que l'ordre s'ha de mantenir a tot l'anell de nodes. Per tant si tenim un anell amb 4 nodes i un rang de calors de l'1 al 100, podríem tenir una estructura així:



Listing 36: Separació en datacenters

On tenim els nodes A, B, C i D. El node A és responsable dels valors de 1 a 25, el B de 26 a 50, el C de 51 a 75 i el D de 76 a 100.

En una taula de Hash distribuïda per buscar un valor, per exemple el de clau 74, el que es farà es el següent:

1. El client vol obtenir el valor associat a la clau 74
2. S'assigna la consulta al node A
3. El node A veu que no és responsable de la clau, per tant envia la consulta al següent node, en sentit horari.
4. El node B fa el mateix que el node A
5. El node C veu que és responsable de la clau i retorna el valor (en cas de tenir-lo)
6. El client rep el valor associat a la clau 74

Aquest procés és el més simple possible i Cassandra (i la majoria de Hash distribuïdes) ofereix vàries optimitzacions al respecte:

- El client coneix quins nodes son responsables de cada clau. Per tant adreça la consulta directament al node adequat.

- Els clients també coneixen quins son els nodes responsables de cada rang. En cas de que els arribi una consulta que no els pertany, la re-envien al node adequat.

Com a qualsevol Hash distribuïda, el més important és definir com es generaran les claus. Ja que això ens determinarà quin serà el rang de les claus, com es distribuiran els rangs pels diferents nodes, i en definitiva, quina serà la distribució de les dades pels diferents nodes.

Cassandra per defecte ofereix dos particionadors, però ens deixa construir els nostres particionadors propis. Aquest és un canvi significatiu respecte Dynamo on únicament existeix un comportament i no es pot canviar. Veiem els dos particionadors que porta Cassandra per defecte.

Particionador per funció de Hash (*RandomPartitioner*)

El primer particionador, és el que fa servir Dynamo, i és una funció de Hash consistent. El que fa aquest particionador és obtenir una clau de Hash a partir de la clau que es vol inserir, i fer servir aquest Hash per a desar la dada. La manera en que es deixen les dades és en ordre ascendent segons la clau de Hash. La funció de Hash oferta és diu que és consistent o forta, que vol dir que s'apropa molt a la distribució perfecta i per tant obtindrem una càrrega gairebé homogènia als diferents nodes.

Aquest particionador és l'ideal per a obtenir una distribució homogènia de les claus. Ens garanteix que si assignem a cada node una part equivalent de l'espai total de claus, la distribució serà gairebé perfecte. Això és ideal per a clústers petits o per a conjunts de dades sense cap patró definit.

Particionador lexicogràfic (*OrderPreservingPartitioner*)

Aquest particionador no efectua cap operació sobre la clau. Simplement fa una ordenació lexicogràfica de les claus tal i com les rep. Aquest particionador no ens ofereix cap tipus de garantia respecte a la distribució de les claus, per tant s'ha d'anar amb compte en fer-lo servir.

La ordenació lexicogràfica és realment útil si es volen obtenir rangs de claus ordenades. Per exemple, si tenim un sistema on desem els missatges d'un usuari, de tal manera que la clau sempre serà "nomUsuari+timeStampï serà molt habitual fer cerques de més d'un missatge a la vegada, com per exemple els 10 últims missatges enviats, aquest tipus de particionador ens serà realment útil. Ja que el més habitual serà que tots els missatges es trobin al mateix node i no només això sinó que en posicions consecutives de disc, per tant amb una sola operació de lectura sobre el disc obtindrem tots els

missatges. Mentre que si féssim servir el particionador aleatori, el més provable seria que haguéssim d'obtenir els missatges de diferents nodes i diferents regions de disc, per tant el temps de consulta seria molt més elevat.

El gran desavantatge del particionador lexicogràfic és que no tenim cap garantia respecte a la distribució de les dades, hem de preocupar-nos nosaltres de proporcionar aquestes garanties, ja sigui o bé usant claus ben pensades per a obtenir aquesta distribució, o bé triant manualment el rang de claus de cada node. En l'exemple anterior, si repartíssim un nombre igual de lletres de l'alfabet a cada node, però tots els usuaris triessin un nom d'usuari començat en "a", tota la càrrega del sistema aniria parar a un mateix node.

La ordenació lexicogràfica és la que fa servir per defecte HBase, mentre que Cassandra fa servir per defecte el particionament aleatori, com es propi de les taules de Hash distribuïdes. No es recomana fer servir el particionador lexicogràfic a no ser que realment sapiguem que és el que necessitarem i les nostres dades s'hi adaptaran perfectament (tindran una bona distribució).

4.3.4 Replicació

Així com a HBase tenim un sistema de fitxers distribuït que és l'encarregat de la replicació, a Cassandra els nodes en si son els encarregats de replicar les dades. Cassandra té un paràmetre configurable que s'anomena el factor de replicació (moltes vegades abreviat com a N), que indica quantes còpies es vol tenir de cada dada.

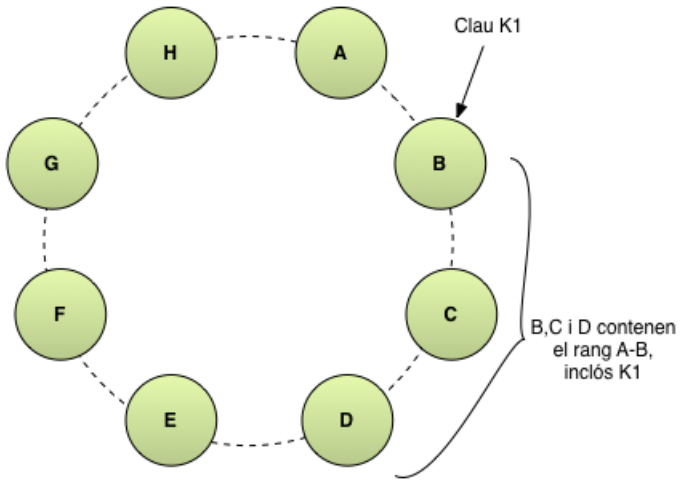
Aquest factor N, determina quantes còpies es vol tenir dels fitxers que contenen les dades. Si tenim N=3, aleshores 3 nodes diferents del sistema tindran una còpia de cada fitxer. De fer la replicació se'n ocupen els nodes, però Cassandra té un paràmetre global que ens indica quin tipus de replicació volem tenir. Com hem vist abans, un clúster pot estar format per varis *datacenters*, i el més habitual és que si tenim varis *datacenters*, vulguem tenir rèpliques de les dades en els diferents *datacenters*, per garantir la disponibilitat de les dades encara que es produeixi una indisponibilitat d'un *datacenter* complert.

Per tant Cassandra ofereix dues tècniques de replicació que van lligades a la topologia del nostre clúster.

Replicació inconscient de l'arquitectura (*RackUnawareStrategy*)

Aquesta opció s'usa si només disposem d'un *datacenter*, o si no volem haver de configurar la topologia. Simplement no té en compte la topologia del clúster i assumeix que es disposa d'un únic *datacenter*.

Aquest tipus de replicació consisteix en una vegada localitzat el node encarregat de la clau que volem inserir, desar-ne una còpia al node corresponent i repartir les $N-1$ còpies restants als $N-1$ nodes següents de l'anell.



Listing 37: Replicació inconscient de l'arquitectura

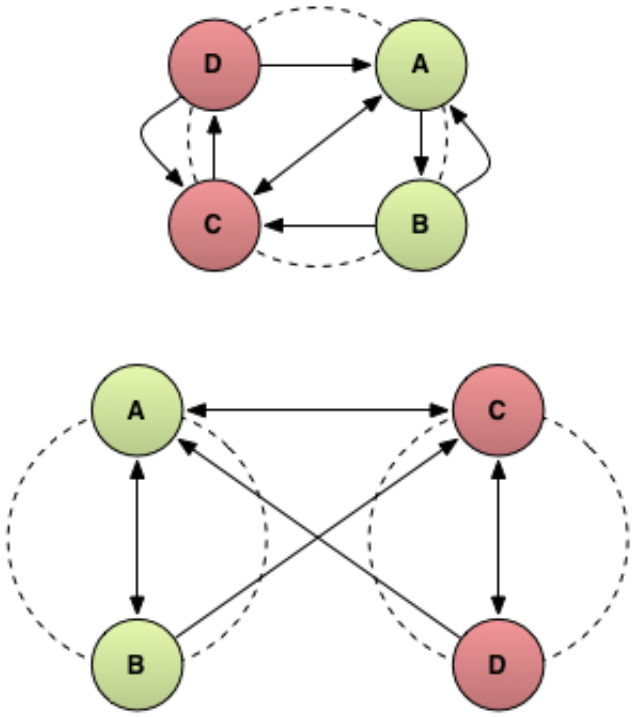
A l'exemple de la figura 37, tenim $N = 3$, i el node B és l'encarregat de la clau que volem inserir, conseqüentment es desa una còpia de la dada als següents nodes del cercle, C i D.

Amb aquesta estratègia de replicació, qualsevol node conté les seves dades i les dels $N-1$ nodes que el precedeixen.

Replicació conscient de l'arquitectura (*RackAwareStrategy*)

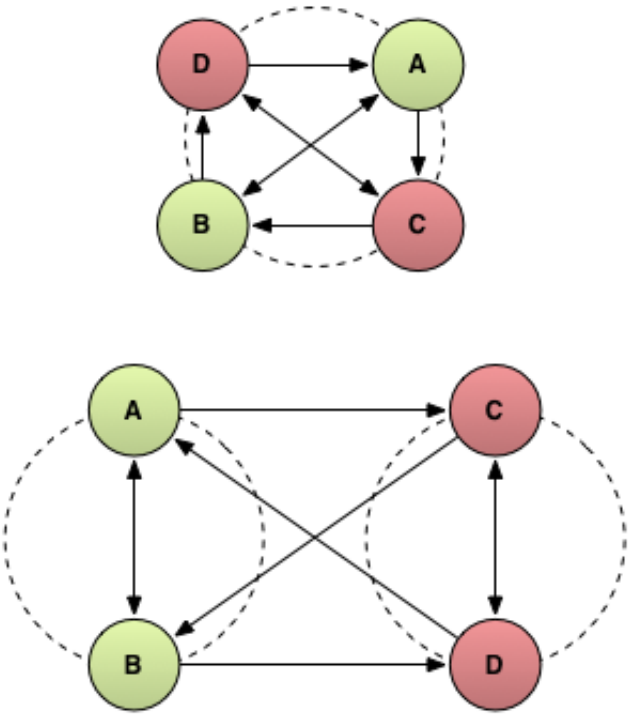
Per usar aquest tipus de replicació cal que el nostre clúster estigui format per més d'un datacenter i que hàgim informat a Cassandra (mitjançant els fitxers de configuració) de la seva topologia.

Aquest estratègia el que fa és al escriure una clau a un node, escriure-la també als següents $N-2$ nodes de l'anell, i escriure'n també una còpia al següent node de l'anell que es trobi a un altre *datacenter*.



Listing 38: Replicació conscient de l'arquitectura

S'ha da tenir en compte en alternar correctament els nodes de diferents *datacenters* a dins l'anell, ja que si per exemple tenim l'estructura de la figura 38 i tenim els nodes A i B a D1, i C i D a DC2, aleshores A i C tindran una quantitat desproporcionada d'informació, ja que tindran la rèplica de tots els nodes de l'altre *datacenter*. Això és degut a que A i C son els primers nodes dels diferents datacenters, i com hem dit l'estratègia de replicació és que tots els nodes d'un datacenter desin una còpia a un node d'un altre datacenter, i amb la topologia de la figura 38 aquests nodes sempre seran A i C.



Listing 39: Replicació conscient de l'arquitectura

Mentre que al cas de la figura 38 el node A tindria les rèpliques de C i D, i C tindria les de A i B. Això passa perquè els següents nodes a l'anell es troben a un *datacenter* diferent, per tant amb l'estratègia de particionament es compleix que com a mínim una còpia ha de'estar a un altre *datacenter*. Per tant si alternem els nodes de l'anell a diferents *datacenters*, la càrrega de cada node queda equilibrada.

Com bé posen a la documentació de Cassandra, el corol·lari és que si es vol afegir un nou *datacenter*, el millor serà que tingui el mateix nombre de nodes que el ja existent.

4.3.5 Operacions

A les operacions de Cassandra és a on podem trobar més similituds amb HBase. Com veurem totes les operacions segueixen pràcticament el mateix esquema. El que sobretot diferencia Cassandra d'HBase, és el fet de no tenir cap element similar al responsable de regió. A Cassandra qualsevol operació es pot realitzar sobre qualsevol node, i en cas de que aquest no pugui realitzar

l'operació sol·licitada (perquè no conté les dades, per exemple), la re-dirigirà al node corresponent i ens retornarà la resposta que obtingui.

Escriptura

El procés d'escriptura segueix el model definit per BigTable, per tant és molt similar al d'HBase. El que canvia més és el fet de que com hem vist, podem triar el nombre d'escriptures a realitzar per garantir o no la consistència. Per tant el valor que posem a W determina el procés d'escriptura.
Per a fer una escriptura Cassandra disposa de 3 elements:

1. commitLog
2. MemTable
3. SSTable

El commitLog és un log on que es desa a disc, que emmagatzema un registre per a cada escriptura realitzada. El commitLog és l'equivalent als dietaris dels SGBDR's. Per altra banda, les MemTables son les estructures de dades que Cassandra fa servir per desar les dades a memòria. Els nodes de Cassandra tenen una MemTable per a cada *ColumnFamily* que contenen. Per últim, les SSTables son els fitxers que Cassandra fa servir per desar les dades a disc.
Passem a veure quin és el camí que segueixen les escriptures a Cassandra:

1. Escriure una entrada al commitLog del node des d'on s'invoca l'escriptura
2. Redirigir l'escriptura als nodes corresponents

Alesohres, per cada node que ha de realitzar una escriptura, se segueixen els següents passos:

1. Escriure una entrada al commitLog del node
2. Escriure la parella clau/valor a la MemCache del node corresponent

Un últim pas, però que no s'executa a cada escriptura és el traspàs de MemCache a un fitxer al disc dur anomenat SSTable (nomenclatura heretada de BigTable). El pas de Memcache a disc és realitza de forma transparent al desenvolupador, a més, és una operació asíncrona i no bloqueja mai cap operació de lectura. A part de les diferents SSTables, també existeix un fitxer d'index anomenat "SSTable Index"que serveix per a localitzar les dades a dins del fitxer SSTable. El procés de passar les dades de la MemCahce a les corresponents SSTables s'activa en vàries circumstàncies:

- S'ha acabat la memòria disponible al sistema.
- Memcache conté masses claus, 128 és la configuració per defecte.
- Cada cert interval de temps (configurable).

Una vegada s'han passat totes les entrades del corresponents al commitLog actual, aquest s'elimina i se'n crea un de nou.
Les propietats de l'escriptura son:

- No es realitza cap lectura
- No s'ha de buscar la posició d'inserció
- Escriptures atòmiques per a cada *columnFamily*
- Sempre es pot escriure

Per tant, Cassandra garanteix que una escriptura no genera cap operació extra, no requereix de cap lectura. Això fa que les escriptures siguin una operació molt ràpida. Un punt a tenir en compte per això és el valor de W seleccionat, ja que si aquest és elevat, tot i que cada escriptura individualment sigui ràpida, el procés d'escriptura es pot relentitzar. Pensant en aquestes situacions, Cassandra ens ofereix dos tipus d'escriptura:

1. *Quorum write*: Es bloqueja el client fins que es rep la confirmació de les W escriptures
2. *Async write*: El client que rep la petició d'escriptura retorna immediatament. Aleshores, el node re-dirigeix les peticions corresponents als nodes corresponents, però el client no ha quedat bloquejat en cap moment.

En qualsevol dels casos, una altra característica del procés d'escriptura de Cassandra, és que si ha de fer una escriptura a un node que no es troba disponible en aquell moment, es fa l'escriptura a qualsevol node i es desa la dada amb una marca que indica al node al que correspon. Cada cert interval de temps s'escaneja el sistema cercant per aquest tipus de dades i es re-distribueixen als nodes corresponents.

Lectura

Les lectures també son molt similars a les d'HBase, però tenen una característica fonamental per al funcionament de Cassandra i l'obtenció de la consistència.

Els passos per a la lectura a un node son:

1. Localització del node que conté la dada
2. Lectura de la dada de la MemCache en cas de que estigui disponible
3. Si la dada no està a MemCache s'obté de la SSTable pertinent
4. Una vegada es té la dada (o conjunt de dades en cas de que estem buscant un rang) es retorna al client

Les lectures es veuen afectades pel valor que hàgim assignat a R. El camí que acabem d'explicar s'ha de fer R vegades, i el client s'ha d'esperar a que s'obtinguin les R respostes. Una vegada es tenen les R respostes, el client rep la dada més actual (en cas de que s'obtinguin diferents versions).

Com hem vist a l'apartat sobre CAP, podria ser que el client rebi una dada inconsistent, una versió que no sigui la més recent de la dada. Això és un risc del que el desenvolupador ha de ser conscient. Per a mirar de disminuir el risc d'inconsistència, Cassandra té el que s'anomena *Read Repair*. El *Read Repair* és un procés que activa Cassandra cada vegada que es fa una lectura, i consisteix en fer $N - R$ lectures de forma asíncrona. És a dir, cada vegada que es llegeix una dada Cassandra l'està sol·licitant sempre a les N rèpliques de forma asíncrona per no bloquejar al client. Aquest procés el que fa és actualitzar ala versió més recent de la dada totes les rèpliques. D'aquesta manera Cassandra s'assegura de que la següent vegada que es faci la lectura d'aquella dada, tots els nodes en tinguin la versió més recent.

Esborrat de dades

Anàlogament al comportament d'HBase, Cassandra no té una opció d'esborrat de dades. El que es fa quan l'usuari vol eliminar una dada es realitzar una escriptura sobre la dada, posant-hi una marca, el que s'anomena *tombstone*. Però així com a HBase s'elimina la dada al fer una compactació, a Cassandra no es pot fer el mateix. Això és degut a que HBase té un coordinador que sap que totes les còpies de la dada s'han marcat com a eliminables, però a Cassandra això no passa degut a que és eventualment consistent, però no es pot assegurar que al moment de fer una compactació la dada que es vol esborrar ha assolit la consistència i per tant se'n esborraran totes les còpies.

Així que el que fa Cassandra és definir un temps anomenat *GCGrace-Seconds*, que normalment és de 10 dies. Com que Cassandra té els "read repairsï altres mecanismes (*AntiEntropy*) per a obtenir la consistència, es considera que es segur eliminar una dada passats 10 dies ja que haurà assolit la consistència. Aleshores al fer la compactació si s'ha sobrepassat aquest *GCGraceSeconds*, la dada serà eliminada.

4.3.6 Fallides i recuperació

Mitjançant el protocol de Gossip, els nodes contínuament van fent operacions de *ping*, cap a els altres nodes. En cas de no rebre resposta s'assumeix la caiguda del node i s'inicia el proces de *Hinted Handoff*.

El procés de *Hinted Handoff* consisteix en que en el moment en que s'assumeix que un node està caigut, es passen rèpliques de les dades destinades a aquest node a les seves rèpliques ja existents més un nou node. Per exemple si tenim els nodes A, B, C, D amb N=3, A, B i C tenen còpies de les dades d'A. Doncs si s'activa *Hinted Handoff*, D passarà a rebre també una còpia de les dades d'A. Amb la particularitat de que a les dades que vagin a parar a D, se'ls posarà una marca que indiqui que les dades anaven destinades al node A. Així, en el moment que el node A es recuperi, el node D li passarà totes les dades que té marcades i les esborrarà.

Aquest *Hinted Handoff* el que implica es que mai perdem la capacitat d'escriure dades, complint l'objectiu de Cassandra que és estar sempre disponible per a rebre escriptures.

La base d'aquest procés és que s'assumeix que un node mai caurà per un llarg temps, es pressuposen parades per manteniment o petites caigudes degudes a fallides d'algun component de Hardware o de comunicació a la xarxa, però es pressuposa que un node sempre tornarà a estar operatiu al futur.

És per això que si volem donar un node com a caigut definitivament, manualment hem d'executar el procés explicat al punt següent "Treure un node".

4.3.7 Afegir i treure nodes

La intenció de Cassandra és oferir la capacitat d'afegir o treure nodes amb total facilitat, veiem els detalls dels dos processos:

Afegir un node *Bootstrapping*

Afegir un node és una operació molt senzilla i totalment automatitzada. Per a fer-ho només ens cal configurar un node per a formar part d'un clúster ja existent i connectar-lo a la xarxa. L'únic paràmetre important a tenir en compte és el rang de claus que volem assignar al nou node. Si el que volem és alliberar càrrega d'un determinat node o volem posar un nou node sabent que farem operacions que carregaran cert node, podem configurar el nou node explícitament per gestiona un rang de claus. Sinó, podem deixar aquesta assignació a Cassandra posant el paràmetre *AutoBootstrap* a la configuració del node.

Si arranquem un nou node amb *AutoBootstrap*, Cassandra buscarà quin és el node que té més càrrega de disc actualment i assignarà la meitat del seu rang de claus al nou node.

Un cop s'afegeix un nou node amb un rang de claus assignat, aquest node es comunicarà amb la resta per fer-los saber de quin rang de claus es farà càrrec, i els nodes que continguin claus que pertanyin al nou node li traspassaran. Una vegada el nou node ha rebut totes les claus (i valors) que li corresponen, ja pot començar a treballar. Aquest procés pot ser bastant llarg ja que la quantitat de dades a rebre pot ser molt elevada. Durant aquest procés el node roman inactiu i desactiva el seu port de consulta per a evitar rebre peticions.

Treure un node

Normalment hi ha dues situacions en les que voldrem treure un node del clúster.

1. Volem retirar un node actiu per qualsevol raó.

2. Un node ha deixat de funcionar i el volem retirar permanentment de clúster.

Per a cada situació s'han de fer servir dos comandes diferents que faran dues operacions diferents:

1. Com que el node que volem retirar encara és funcional, es retiraran les dades d'aquest node abans de permetre que l'enretirem. Un cop ja s'en han extret les dades, es notifica a la resta de nodes que aquest ha deixat de ser operatiu.

2. En cas de que el node que s'encarregava d'un rang de claus deixi de funcionar, no en podem extreure la informació, per tant la informació

s'extraurà de la resta de rèpliques existents i després es notificarà a la resta de nodes de que el node ja no existeix.

Si afegim o traiem nodes, correm el disc de que acabem amb un clúster mal balancejat, amb una distribució no homogènia. Per tant el que es recomana és re-assignar a cada node un rang de claus que resulti en un clúster més ben balancejat. Existeixen alguns algorismes que ens donen la millor distribució del rang de claus per a cada node.

4.4 Model de dades

El model de dades és heretat de BigTable, per tant, conceptualment similar al d'HBase. Un model de dades orientat a columnes i sense necessitat de definir un esquema per a les columnes. Però a la pràctica té dos canvis importants, les superColumnes, i la ordenació. Anem a veure una descripció dels diferents elements que composen el model de dades i per últim veurem com funciona la ordenació dins d'aquest model.

Primer cal dir que l'element que ho engloba tot s'anomena *keyspace*, que vindria a ser el nom de la base de dades. Així com a HBase podem tenir varies bases de dades, a Cassandra només podem tenir un únic *keyspace*.

Totes les explicacions que es faran a aquest capítol apliquen per a la versió 0.6 de Cassandra. A més durant tot el capítol farem servir una notació diferent a la que s'ha fet servir al capítol d'HBase. Aquí totes les estructures de dades es descriuran en notació estil JSON, ja que a Cassandra és la notació que es fa servir.

4.4.1 Columnes (*Column)*

Una columna és exactament el mateix que a HBase, una columna disposa de 3 elements, el nom de la columna, el seu valor i un timestamp. Per tant Cassandra també fa servir versionat de les dades. Així una columna seria:

```
{
    nom: "email",
    valor: "test@example.com",
    timestamp: 123456789
}
```

Listing 40: Exemple de Columna

És important notar que els camps nom i valor son binaris, en concret son arrays de bytes (byte[]).

4.4.2 SuperColumnes (*SuperColumn*)

Una supercolumna és un mapa de columnes, que pot contenir un nombre indefinit de columnes. Les supercolumnes s'identifiquen per una clau, en aquest cas el seu nom.

```
{
    nom: "casa",
    valor: {
        carrer: {
            nom: "carrer",
            valor: "carrer major",
            timestamp: 123456789},
        ciutat: {
            nom: "ciutat",
            valor: "Barcelona",
            timestamp: 123456789},
        zip: {
            nom: "zip",
            valor: "08154",
            timestamp: 123456789},
    }
}
```

Listing 41: Exemple de SuperColumna

A aquest exemple casa és una super columna, i carrer, ciutat i zip en son les seves columnes. D'ara en endavant simplificarem la notació. En la nova notació l'exemple anterior seria:

```
"casa" : {
    carrer: "carrer major"
    ciutat: "Barcelona"
    zip: "08154"
}
```

Listing 42: Notació simplificada per a les SuperColumnes

4.4.3 Família de columnes (*ColumnFamily)*

El concepte de família de columnes és el mateix que a HBase, però amb la diferència que al igual com pasa amb les columnes, una família de columnes també pot ser *super*

ColumnFamily

Al igual que a HBase una família de columnes, conté files. Una fila s'identifica per una clau i està formada per un nombre indefinit de columnes.

```
Usuaris: {
    Joan: {
        nom: "Joan Perez",
        email: "Joan@example.com",
        edat: "25"
    },
    Maria: {
        nom: "Maria Gomez",
        email: "maria@example.com",
        edat: "23",
        telefon: "931234567"
    }
}
```

Listing 43: Exemple de Família de Columnes

A l'exemple de la figura 43 la família de columnes és Usuaris, i les files estan identificades per les seves claus Joan i Maria. Com podem veure el nombre de columnes que conté cada fila és variable, no està definit, és el que es diu sense esquema (*SchemaLess*).

SuperColumnFamily

Una Super família de columnes és l'equivalent a una família de columnes però usant superColumnes en comptes de columnes normals.

```
Agenda: {
    escola: {
        Joan: {
            nom: "Joan Perez",
            email: "Joan@example.com",
            edat: "25"
        },
        Maria: {
            nom: "Maria Gomez",
            email: "maria@example.com",
            edat: "23",
            telefon: "931234567"}
    },
    feina: {
        Marc: {
            nom: "Marc Ruiz",
            email: "marc@example.com",
            edat: "28"
        },
    }
}
```

Listing 44: Exemple de Super Família de Columnes

Aquí la *SuperColumnFamily* és Agenda, les claus d'aquesta son les seccions de l'agenda (escola i feina). A dins de cada *SuperColumnFamily* hi tenim *SuperColumns* que son els noms dels contactes (Joan, Maria, etc) que a la seva vegada tenen diferents columnes (nom, edat, etc.). El nombre de *SuperColumns* és il·limitat així com el nombre de columnes a cada una d'aquestes.

4.4.4 Ordenació

Cassandra té una característica que no té HBase que és la ordenació de les columnes. Aquesta ordenació es pot realitzar a dos nivells, a les columnes i a les superColumnes. La ordenació sempre es realitza segons el nom, el que es pot canviar és l'algorisme d'ordenació. Per defecte tenim ordenació lexicogràfica, ordenació per a dates, per a *Longs*, etc. Però Cassandra també ens permet implementar els nostres propis algorismes d'ordenació.

Veiem primer com és la ordenació a nivell de columnes. Si tenim les següents columnes:

```
{nom: 3, valor: "123"}
{nom: 123, valor:"hola"}
{nom: 45, valor:"456"}
```

Listing 45: Columnes d'exemple

Si apliquem ordenació del tipus *LongType*, les columnes quedarien ordenades de la següent manera:

```
{nom: 3, valor: "123"}
{nom: 45, valor:"456"}
{nom: 123, valor:"hola"}
```

Listing 46: Columnes ordenades com a *LongType*

En canvi si apliquem ordenació del tipus *UTF8Type* (ordenació lexicogràfica amb codificació UTF8):

```
{nom: 123, valor:"hola"}
{nom: 3, valor: "123"}
{nom: 45, valor:"456"}
```

Listing 47: Columnes ordenades com a *UTF8Type*

El següent pas és la ordenació segons la SuperColumna. Aquesta ordenació no és exclusiva, és a dir, podem ordenar a la vegada per SuperColumna i per Columna. Veiem-ne un exemple:

```
Marc: {
    nom: "Marc Ruiz",
    email: "marc@example.com",
    edat: "28"
},
Joan: {
    nom: "Joan Perez",
    email: "Joan@example.com",
    edat: "25"
},
Maria: {
    nom: "Maria Gomez",
    email: "maria@example.com",
    edat: "23",
    telefon: "931234567"
}
```

Listing 48: SuperColumnes d'exemple

Si a aquesta estructura apliquem ordenació *UTF8Type* tant a les supercolumnes com a les columnes, obtenim:

```
Joan: {
    edat: "25",
    email: "Joan@example.com",
    nom: "Joan Perez"
},
Marc: {
    edat: "28",
    email: "marc@example.com",
    nom: "Marc Ruiz"
},
Maria: {
    edat: "23",
    email: "maria@example.com",
nom: "Maria Gomez",
    telefon: "931234567"
}
```

Listing 49: SuperColumnes d'exemple

No cal que les dues ordenacions aplicades siguin les mateixes, podríem aplicar per exemple *UTF8Type* a nivell de SuperColumna i *LongType* a nivell de columna. Qualsevol combinació és possible.

4.5 Conclusions

Cassandra proposa un nou model a l'hora d'implementar el model proposat per BigTable. Aquesta nova implementació usant una taula de Hash distribuïda, té punts forts i punts on HBase és més robust, però Cassandra és una eina molt potent i amb molta bona acollida per part dels usuaris.

Els punts forts de Cassandra son:

- No depen de cap altra eina, instal·lant Cassandra ja tenim tot el que necessitem
- No té cap element central, podem realitzar qualsevol operació sobre qualsevol node
- Escriptures sempre disponibles. Encara que el node responsable de la dada estigui inactiu, l'escriptura es realitza
- Gran capacitat de configuració. La gran majoria de paràmetres es poden modificar, permetent adaptar l'eina a les necessitats de l'aplicació

Tot i que alguns d'aquests punts poden ser vistos com punts dèbils per alguns. La majoria de les queixes en torn a Cassandra es centren en que és massa complicat aconseguir una bona configuració. O el fet de no tenir una estratègica única en front del teorema CAP, sent l'usuari qui ha de decidir aquests paràmetres.

Al contrari que HBase, Cassandra va ser adoptada des de l'inici per a usos més interactius, per a ser usada com a base de dades real d'aplicacions. Tot i que ràpidament es va veure que calia oferir integració amb eines com Hadoop per poder assolir una quota de mercat més elevada i amb les últimes versions de Cassandra [3], aquesta integració comença a ser sòlida i ja hi ha empreses que també estan usant Cassandra per emmagatzemar els seus logs. Cassandra en aquest aspecte ofereix escriptures més ràpides (a priori) que HBase, sobretot usant $W = 1$, lo qual és més que suficient per a desar logs, ja que son dades molt poc interactives i el més provables es que només es llegeixin directament des de Hadoop.

Des del meu punt de vista, Cassandra és una eina molt potent, i el fet de poder triar el nivell de consistència adequat per a cada aplicació o inclús per a diferents tipus d'operacions, dona una potència enorme a Cassandra. Segurament no és l'eina més indicada si la Consistència de les dades és un punt crucial per a les nostres aplicacions, ja que segurament això ens farà tenir una aplicació lenta. Però si la nostra aplicació ens permet treballar amb

[3] a partir de la 0.7

$W = 1$ o $R = 1$, Cassandra ens ofereix una eina molt ràpida i apte per a la gran majoria d'aplicacions actuals.

Tot i que Cassandra requereix més atenció a la configuració per part del desenvolupador, crec que és un risc que val la pena assumir, ja que una vegada s'està familiaritzat amb Cassandra, aquest grau de flexibilitat acaba sent un punt a favor.

Si l'èxit d'un sistema es mesurés per la quantitat d'articles que en parlen a internet i el volum de la seva comunitat d'usuaris, es por dir que durant el 2010 Cassnadra ha estat l'eina NoSQL més exitosa. Tot i que a l'últim trimestre de l'any els seus creadors, Facebook, van donar anunciar que estan construint dos sistemes a sobre d'HBase, el seu sistema de logging anomenat Scribe, i la seva nova plataforma de missatgeria. Per tant una part del focus que tenia Cassandra, ara ha tornat a HBase, però la comunitat que usa Cassandra segueix sent molt gran, i sembla que segueix creixent.

Capítol 5

Primer cas pràctic

El primer cas pràctic serveix per a fer un primer contacte amb els dos entorns HBase i Cassandra. Com que al començar el projecte desconeixia totalment els dos entorns, amb aquest primer cas pràctic el que em vaig plantejar és familiaritzar-me amb els dos entorns, per tal d'entendre millor els dos productes, posar en pràctica els coneixements assolits i plantejar el segon cas pràctic.

A nivell de coneixements, al realitzar aquest cas pràctic no tenia tots els coneixements que hi ha reflectits en els dos capítols anteriors. De fet, la gran majoria dels materials en que es basen els dos capítols anteriors, han estat creats, o els he trobat, després d'haver realitzat aquest cas pràctic. Per tant moltes de les característiques (i limitacions) dels productes les he trobat programant i no mirant la documentació. Això pot ser un error degut a que em vaig precipitar al fer aquest primer cas, però en realitat crec que el fet de trobar aquestes limitacions i diferencies em va portar a voler-me documentar millor i així aconseguir l'objectiu de començar el segon cas pràctic amb un molt bon coneixement de les dues plataformes.

Aquest primer cas pràctic consisteix en provar de fer dues versions d'una mateixa aplicació, cadascuna fent servir una base de dades diferent. Com que tampoc era l'objectiu desenvolupar una aplicació des de zero, vaig optar per començar amb una aplicació ja existent desenvolupada per a Cassandra. Primer vaig estudiar com estava desenvolupada aquesta aplicació, quines eren les seves parts i com era la integració amb Cassandra. Una vegada vaig conèixer bé l'aplicació vaig migrar la capa de connexió amb Cassandra per a que passés a funcionar amb HBase.

5.1 Twissandra

Com que no és l'objectiu d'aquest projecte el desenvolupar cap gran aplicació, sinó estudiar la integració amb les bases de dades NoSQL, vaig decidir partir d'una aplicació ja existent. A la documentació de Cassandra, ofereixen una aplicació com a exemple per entendre millor el funcionament d'aquesta. L'aplicació en qüestió s'anomena Twissandra, i consisteix en una adaptació a petita escala de Twitter.

Al explorar una mica l'aplicació vaig veure que s'adaptava perfectament a les meves necessitats per 3 raons:

1. El funcionament de Twitter és molt senzill i conegut per tothom (o molt fàcil d'explicar).

2. Ben documentada i codi molt fàcil d'entendre.

3. Separació de la capa de comunicació amb la base de dades.

Dels punts anteriors el més important és el tercer. Tota la comunicació que fa Twissandra amb Cassandra està contingut en un sol fitxer. Això facilita molt la feina, ja que només m'he de preocupar d'entendre el contingut d'aquest fitxer despreocupant-me de la resta, que no entra dins l'abast d'aquest projecte. A part, em facilitava molt la migració a HBase ja que només modificant aquest fitxer, ja tenia exactament la mateixa aplicació funcionant amb HBase.

Així que una vegada vaig veure que Twissandra era l'aplicació adient, vaig començar a analitzar les seves diferents parts.

5.1.1 Estructura general

Twissandra és una aplicació web desenvolupada en Djago. Django és un *framework* de Python per al desenvolupament d'aplicacions web. Per tant tots els fragments de codi mostrats en aquest apartat es mostraran en aquest llenguatge de programació. Per als objectius d'aquest apartat no cal que sapiguem res més sobre Django.

Twissandra com a aplicació emula Twitter. Twitter és el que s'anomena una xarxa social de *micro-blogging*. La seva part principal i la que emula Twissandra està formada per dues funcionalitats principals:

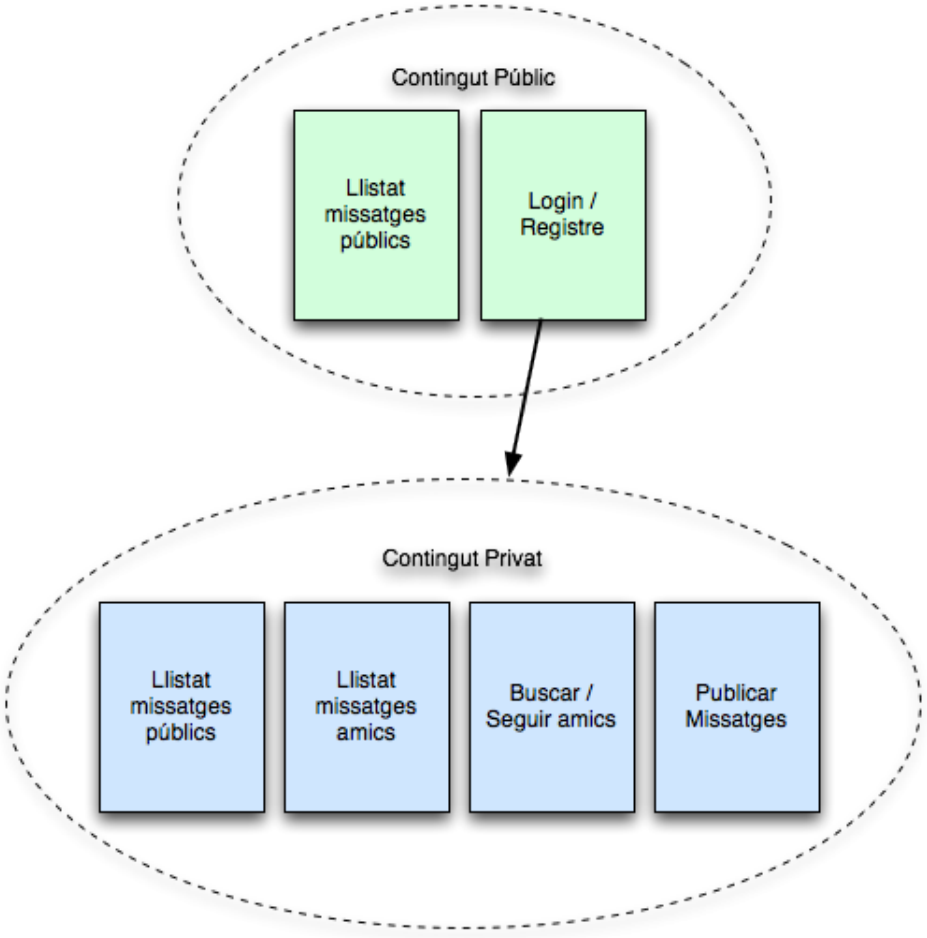
1. Publicar missatges

2. Seguir usuaris

La primera i més bàsica consisteix en la publicació de missatges curts, amb un màxim de 140 caràcters. Per això es diu *micro-blogging*. La segona funcionalitat consisteix en definir relacions entre els usuaris. Un usuari A pot seguir a qualsevol altre usuari B. Això crea automàticament dues relacions: A segueix a B, i B és seguit per A. El resultat d'aquesta operació es limita a que un usuari rebrà tots els missatges que publiquin els usuaris als que segueix. Per tant si A segueix a B, C i D, cada vegada que un d'aquests publiqui un missatge, aquest apareixerà a la pàgina principal d'A.

Aques és un esquema una mica simplista del Twitter actual, però aquestes son les seves funcionalitats centrals i son les que ofereix Twissandra.

Twissandra, al igual que Twitter té una part pública visible per tothom, i una part privada només disponible per als usuaris registrats.



Listing 50: Agrupació de pàgines segons accessibilitat

En resum, qualsevol persona pot veure els continguts que es publiquen a Twissandra, però per a poder generar contingut o seguir a usuaris cal estar registrat.

5.1.2 Model de dades

El model de dades que fa servir Twissandra és molt senzill i es divideix en 7 families de columnes (taules) diferents.

User

La clau de cada fila és un *userId* únic per a cada usuari, i les columnes son les diferents propietats de cada usuari (id, username, password, etc.).

```
User = {
    'a4a70900-24e1-11df-8924-001ff3591711': {
        'id': 'a4a70900-24e1-11df-8924-001ff3591711',
        'username': 'joan',
        'password': '****'
    }
}
```

Listing 51: Model de dades per als usuaris

Aquesta taula fa servir com a comparador *UTF8Type*. El comparador serveix per a ordenar les columnes quan es fan operacions de *slicing* (cerca de rangs). Així, per exemple, si es fa una cerca de 10 usuaris, els podem ordenar segons la columna nom.

Username

Aquesta és una taula específica per a quan es fan consultes per nom d'usuari en comptes de per id. La clau és el nom d'usuari i la única columna que té és l'id d'aquest usuari. Com veurem a la resta de taules, tota la informació es desa amb l'id d'usuari, però hi haurà certes operacions que permetran fer consultes per nom d'usuari. Aquesta taula únicament serveix per a poder fer la conversió entre un nom d'usuari i el seu id correponent.

```
Username = {
    'joan': {
        'id': 'a4a70900-24e1-11df-8924-001ff3591711'
    }
}
```

Listing 52: Model de dades per als noms d'usuaris

Aquesta taula fa servir el comparador *BytesType*. A aquesta taula mai hi farem operacions de cerca múltiple per tant el comparador és irrellevant.

Friends i Followers

Aquestes dues taules serveixen per a guardar les relacions entre usuaris. Tan els usuaris als que seguim, com els usuaris que ens segueixen. L'estructura de les dues taules és idèntica, la clau de la fila és l'id d'un usuari, i el contingut és una columna per a cada usuari al que es segueix (o ens segueix) formada pel seu id i un *timestamp* que indica quan es va crear la relació. Aquest *timestamp* únicament es desa ja que és informació útil per a ser mostrada (ex: segueixes a aquest usuari des d'el 24-03-2010).

```
Friends = {
    'a4a70900-24e1-11df-8924-001ff3591711': {
        '10cf667c-24e2-11df-8924-001ff3591711': '1267413962580791',
        '343d5db2-24e2-11df-8924-001ff3591711': '1267413990076949',
        '3f22b5f6-24e2-11df-8924-001ff3591711': '1267414008133277'
    }
}

Followers = {
    'a4a70900-24e1-11df-8924-001ff3591711': {
        '10cf667c-24e2-11df-8924-001ff3591711': '1267413962580791',
        '343d5db2-24e2-11df-8924-001ff3591711': '1267413990076949',
        '3f22b5f6-24e2-11df-8924-001ff3591711': '1267414008133277'
    }
}
```

Listing 53: Friends i Followers: Relacions entre usuaris

Aquestes taules fan servir el comparador *BytesType*. A aquestes taules tampoc ens importa el comparador ja que mai mostrarem els resultats de la cerca sobre aquestes taules.

Timeline i Userline

Aquestes dues taules serveixen per a crear els dos llistats de missatges anteriorment explicats. Timeline desa tots els missatges generats al sistema, agrupats per l'id de l'usuari que els genera. A la taula timeline hi ha un id especial que es fa servir per a desar tots els tweets que s'insereixen al sistema. La taula userline serveix per a deixar els tweets que van dirigits a cada usuari, és a dir, a la fila corresponent a cada usuari s'afegeixen tots els tweets que insereixen els usuaris als que ell segueix.

```
Timeline = {
    'a4a70900-24e1-11df-8924-001ff3591711': {
        # timestamp of tweet: tweet id
        1267414247561777: '7561a442-24e2-11df-8924-001ff3591711',
        1267414277402340: 'f0c8d718-24e2-11df-8924-001ff3591711',
        1267414305866969: 'f9e6d804-24e2-11df-8924-001ff3591711',
        1267414319522925: '02ccb5ec-24e3-11df-8924-001ff3591711'
    }
}

Userline = {
    'a4a70900-24e1-11df-8924-001ff3591711': {
        # timestamp of tweet: tweet id
        1267414247561777: '7561a442-24e2-11df-8924-001ff3591711',
        1267414277402340: 'f0c8d718-24e2-11df-8924-001ff3591711',
        1267414305866969: 'f9e6d804-24e2-11df-8924-001ff3591711',
        1267414319522925: '02ccb5ec-24e3-11df-8924-001ff3591711'
    }
}
```

Listing 54: Timeline i Userline: classificació dels missatges publicats

Aquestes dues taules fan servir el comparador *LongType*. Això ens serveix per a fer l'ordenació sobre la columna del *timestamp*. Així podem seleccionar per exemple, els 10 últimes *tweets* publicats.

API Cassandra

Cassandra està desenvolupat íntegrament en Java, però curiosament per accedir a l'API de Cassandra no existeix un client natiu en Java. Per accedir a Cassandra, en el llenguatge que sigui, hem de fer-ho a través de Thrift.

Les crides disponibles mitjançant Trift per Cassandra son:

Cass.py

Fent ús de l'API que acabem de descriure, Twissandra implementa totes les consultes a un únic fitxer anomenat cass.py. Ara en veurem les parts més rellevants.

El primer que conté el fitxer és la definició de totes les funcions que implementarà. No les implementacions d'aquestes, sinó únicament un llistat amb els seus noms. Les funcions que es faran servir per l'aplicació en si son:

Nom Consulta	Paràmetres	Operació
get_user_by_id	user_id	Donat un id retorna l'usuari corresponent
get_user_by_username	username	Donat un nom d'usuari retorna l'usuari corresponent
get_friend_ids	user_id	Donat un id d'usuari retorna una llista amb ids dels usuaris als que segueix
get_follower_ids	user_id	Donat un id d'usuari retorna una llista amb ids dels usuaris que el segueixen
get_users_for_user_ids	user_ids	Donada una llista d'ids d'usuari, retorna una llista amb els usuaris associats
get_friends	user_id	Donat un id d'usuari, retorna un conjunt amb tots els usuaris als que segueix
get_followers	user_id	Donat un id d'usuari, retorna un conjunt amb tots els usuaris que el segueixen
get_timeline	user_id	Donat un id d'usuari, retorna el seu *timeline* (missatges dels usuaris als que segueix)
get_userline	user_id	Donat un id d'usuari, retorna el seu *userline* (els seus missatges)
get_tweet	tweet_id	Donat un id de missatge, retorna el missatge
save_user	user_id, user	Desa una nova instància d'usuari
save_tweet	tweet_id, user_id, tweet	Desa una nova instància de missatge
add_friends	from_user, to_users	Donat un id d'usuari A i una llista d'ids d'usuaris. Crea les relacions d'amistat (friend + follower) Entre A i tots els id's de la llista
remove_friends	from_user, to_users	Donat un id d'usuari A i una llista d'ids d'usuaris. Elimina les relacions d'amistat entre A i els usuaris de la llista

Listing 55: Funcions de comunicació amb la base de dades

A part de les funcions definides a la figura 55, n'hi ha dues més que son considerades privades i per això no estan a la definició inicial. Aques-

tes dues funcions serveixen per estructurar el codi i fer-lo més reusable, en concret, en tenim una que es fa servir per a fer *get_followers* i *get_friends* que es diu *_get_friend_or_follower_ids*. Aquesta funció retorna una llista de *friends* o *followers* segons des d'on es cridi. La segona funció és la que criden *get_timeline* o *get_userline* i s'anomena *_get_line*. Aquesta funció simplement retorna una llista de tweets, que anirà a buscar a una *column family* o a una altra depenent des d'on es faci la crida.

El següent que es defineix a cass.py és el mapeig amb totes les taules de la base de dades:

```
CLIENT = pycassa.connect_thread_local(framed_transport=True)
USER = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'User',
ct_class=OrderedDict)
USERNAME = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'Username',
        dict_class=OrderedDict)
FRIENDS = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'Friends',
        dict_class=OrderedDict)
FOLLOWERS = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'Followers',
        dict_class=OrderedDict)
TWEET = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'Tweet',
        dict_class=OrderedDict)
TIMELINE = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'Timeline',
        dict_class=OrderedDict)
USERLINE = pycassa.ColumnFamily(CLIENT, 'Twissandra', 'Userline',
        dict_class=OrderedDict)
```

Listing 56: Mapeig de les taules de Cassandra

Com hem vist anteriorment, per a connectar-nos amb Cassandra ho hem de fer a través de Thrift, per a facilitar-nos la feina, existeix una llibreria anomenada pycasa que ens facilita la feina encarregant-se de tots els paràmetres de configuració. Així, usant pycasa, simplement hem d'implementar les consultes i la llibreria s'encarrega de la resta.

A la figura 56 podem veure com es fa el mapeig de les taules de Cassandra a l'aplicació. La primera línia defineix la connexió amb la base de dades a través de pycassa. La resta de columnes simplement s'encarreguen de mapejar les taules de la base de dades. Com podem veure per a mapejar una taula ho fem amb l'objecte CLIENT, que és el que té la connexió amb la bd, i li passem el nom del *keyspace* (Twissandra), i el nom de la taula que volem mapejar. Per últim s'especifica el tipus d'ordenació que fa servir la família de columnes.

Una vegada realitzat amb el mapeig amb la base de dades, es defineix

una constant

```
PUBLIC_USERLINE_KEY = '!PUBLIC!'
```

Aquesta constant es farà servir per a inserir tots els tweets que es publiquin a Twissandra al *Userline* d'un usuari public. D'aquesta manera s'aconsegueix poder tenir una vista amb els últims tweets publicats al sistema.

Per últim trobem la implementació de les funcions. En posem una d'exemple per a mostrar-ne el funcionament, però entrarem en més detall a les funcions als pròxims apartats.

```
1  def get_user_by_id(user_id):
2   """
3   Given a user id, this gets the user record.
4   """
5   try:
6               user = USER.get(str(user_id))
7   except NotFoundException:
8               raise NotFound('User %s not found' % (user_id,))
9   return user
```

Listing 57: Implementació de la funció get_user_by_id

Com es pot veure a la figura 57, el codi per a buscar un usuari és bàsicament una línia, la numero 6, a on es fa servir l'objecte de mapeig amb la base de dades *USER* per a realitzar un simple get. Al ser una base de dades clau:valor, amb la clau de l'usuari (l'id) obtenim tots els seus valors. La resta de la funció és la seva descripció, linies 2-4 i la gestió d'errors, línies 5 i 7-8

Limitacions de Twissandra

Twissandra té una clara limitació que inclús explica el seu autor al fitxer cass.py. Aquesta limitació la trobem al fet d'emmagatzemar tots els tweets sota el nom d'usuari definit per la constant PUBLIC_USERLINE_KEY. La versió de Cassandra 0.6 té una limitació i és que requereix que tot el contingut d'una fila es pugui encabir a memòria. Això és clarament una limitació ja que si el sistema tingués una càrrega alta, en algun punt es superaria aquesta limitació. La solució que proposa l'autor és generar claus públiques

combinades amb *timestamps*. És a dir, tenir una clau pública per a cada dia, o inclús per a cada hora. D'aquesta manera ens seria molt més difícil sobrepassar el limit imposat per Cassandra. Amb aquesta modificació la clau passaria a ser, per exemple, per al dia 1 de Gener de 2010:

```
PUBLIC_USERLINE_KEY = '!PUBLIC!01-01-2010'
```

Aquesta modificació suposa varis canvis al codi de Twissandra, i com que únicament es tracta d'una aplicació de test i complicaria la lectura del codi per a usuaris no avançats, l'autor decideix no realitzar aquests canvis.

Com a nota vull indicar que amb la versió 0.7 de Cassandra la limitació de mida de fila, passa a ser l'espai disponible en disc en comptes de l'espai disponible a memòria.

5.2 Twitbase

Twitbase és una adaptació de Twissandra per a fer servir HBase en comptes de Cassandra. La idea inicial era únicament modificar les funcions de consulta a la base de dades explicades a l'apartat anterior, i passar a usar HBase. El que a priori era una tasca molt simple, va acabar no sent-ho tant, i la versió final de Twitbase que es mostra en aquest apartat té canvis inclús a nivell de model de dades per tal d'oferir un bon funcionament.

5.2.1 Canvis respecte Twissandra

De Pycassa a Thrift

Així com Twissandra deixa totes les tasques de connexió a la base de dades a una llibreria anomenada Pycassa, que fa d'enllaç amb les crides a Thrift, no es va creure necessari buscar una eina similar i es va passar a treballar directament amb la llibreria Trhift. A priori la majoria de crides de Pycassa son simples encapsulacions de les crides que ofereix Trhift, per tant eliminar aquesta eina intermitja no havia de suposar massa problemes.

Aquest canvi, òbviament implica haver d'afegir una mica més de codi cada vegada que es vol fer una connexió amb la base de dades. Mentre que amb Twissandra aquesta connexió es feia amb poc codi:

```
import pycassa

CLIENT = pycassa.connect_thread_local(framed_transport=True)
```

Listing 58: Connexió amb Cassandra fent servir pycassa

Per a fer el mateix però connectant a HBase directament amb Thrift, el codi passa a ser més llarg:

```
from thrift import Thrift
from thrift.transport import TSocket
from thrift.transport import TTransport
from thrift.protocol import TBinaryProtocol
from hbase import Hbase
from hbase.ttypes import *

# Make socket
transport = TSocket.TSocket('localhost', 9090)
# Buffering is critical. Raw sockets are very slow
transport = TTransport.TBufferedTransport(transport)
# Wrap in a protocol
protocol = TBinaryProtocol.TBinaryProtocol(transport)
client = Hbase.Client(protocol)
transport.open()
```

Listing 59: Connexió amb HBase directament amb Thrift

Aquest codi es troba a tots els fitxers que es mostren a aquest capítol i treballen sobre HBase, però no tornaré a mostrar-lo per facilitar la lectura del codi.

Canvis al model de dades

Per a fer Twitbase he realitzat dos al model respecte de l'original, Twissandra. Un dels canvis va estar premeditat, però l'altre va sorgir a mida que anava desenvolupant l'aplicació.

El primer canvi és l'eliminació de la taula *Username*. Des de que vaig començar a fer servir Twissandra no entenia el perquè d'aquesta taula. El seu únic objectiu semblava ser fer una traducció del nom d'usuari a un id únic d'usuari que es generava amb un algorisme de codificació. Però des del meu punt de vista, no té cap sentit codificar el nom d'usuari i treure'n un id, bàsicament perquè l'objectiu d'aquesta operació l'únic que fa és complicar

l'estructura de l'aplicació, i dificultar la lectura de les dades a la base de dades. El fet de codificar les dades normalment es fa per a evitar que les dades puguin ser llegides, o per a buscar una codificació més adient per desar una dada a la base de dades, però en el cas del nom d'usuari em va semblar no només irrellevant sinó incòmode. Per tant, des d'un inici, Twitbase no va comptar amb la taula *Username*.

Com a nota m'agradaria dir que les últimes versions de Twissandra també han fet aquest canvi i ja no tenen la taula *Username*.

El segon canvi va sorgir a mida que anava fent el desenvolupament. Al afegir o treure amics, s'havien de fer operacions sobre dues taules, la de *Friends* i la de *Followers*, però a mida que anava desenvolupant, em semblava una mica incongruent fer servir aquestes taules. Al model relacional, l'ús d'aquesta taula és obvi, ja que és una relació molts a molts, i no hi ha una altra manera de fer-ho. Això és degut a que un usuari pot tenir molts amics i pot ser seguit per moltes persones. Com que els sistemes relacionals fan servir models de dades estàtics, però aquestes relacions son dinàmiques, no hi ha altra manera de mapejar aquesta relació que no sigui fent servir una taula.

Però això ja no passa amb els sistemes NoSQL. Als sistemes NoSQL els esquemes son dinàmics, i podem afegir columnes a mida que ens facin falta, per tant sembla més coherent amb el model de dades dels sistemes NoSQL, usar columnes en comptes de taules per a desar les relacions entre usuaris. De fet, és el que està fent el model de dades de Twissandra, ja que tots els amics i seguidors es desen a una columna identificada per un id d'usuari.

En resum les taules *Friends* i *Followers*, simplement serveixen per a separar la taula usuaris en múltiples taules. Per tant no veia el sentit de matenir aquesta estructura i Twitbase no té les taules *Followers* ni *Friends*, sinó que aquesta informació passa a estar en dues columnes de la taula *Users* .

De la mateixa manera com semblava adient fer aquest canvi amb *Friends* i *Followers*, també hagués tingut sentit fer-ho amb Timeline i Userline, ja que la clau de ambdues taules també és l'id de l'usuari. Però per precaució només vaig fer el canvi mostrat i l'altre el vaig deixar per quan hagués comprovat que realment era factible i recomanable fer-ho en columnes en comptes d'amb taules.

Al haver fet aquest canvi, em vaig adonar del possible problema que suposa aquesta estructura de dades. Si posem tots els seguidors en una única columna de la fila corresponent a cada usuari, podem topar amb el mateix límit que trobem a Twissandra al deixar tots els *Tweets* públics sota un mateix nom d'usuari. El problema és que una fila és la mínima quantitat de dades disponible, una fila és indivisible. Com hem vist a Twissandra aquest límit es tradueix en que una fila no pot ser més gran que l'espai disponible

en un únic disc dur, si creix més d'això el sistema no sabrà com manejar-ho i donarà error. Per això s'ha d'anar en compte amb les mides de les files. Però presumiblement un usuari mai tindrà suficients amics o seguidors com per a poder forçar aquest error.

Més aviat aquesta limitació sembla més que vingui imposada per la manera de pensar a la que estic acostumat després de treballar amb el model relacional que per HBase en si, ja que de fet aquest tipus d'estructures de dades és el que se suposa que busca HBase, milions de files amb milions de columnes. Per tant vaig decidir fer el canvi només amb *Friends* i *Followers* i veure si em trobava amb algun tipus de problema.

5.2.2 createDB.py

El primer que vaig fer abans de començar Twitbase va ser crear un petit script que creés les taules a HBase. Per defecte HBase ve configurat per a desar les dades al directori /tmp del sistema operatiu, per tant les dades es perden al apagar el servidor. Tot i que és trivial canviar aquest directori i que les dades siguin realment persistents, vaig decidir deixar-ho així i crear l'estructura cada vegada i començar amb la base de dades buida. Per això em va ser realment útil tenir un script per inicialitzar la base de dades.

```
def crearTaula(nom, cols):
    try:
        client.createTable(nom, cols)
    except AlreadyExists, tx:
print "La taula %s, ja existeix." % (nom)

def crearTaules():
    print client.getTableNames()
    cols = []
    cols.append(ColumnDescriptor( name='info' ))
    cols.append(ColumnDescriptor( name='friends' ))
    cols.append(ColumnDescriptor( name='followers' ))
    crearTaula('User', cols)
    cols = []
    cols.append(ColumnDescriptor( name='user_id' ))
    cols.append(ColumnDescriptor( name='id' ))
    cols.append(ColumnDescriptor( name='body' ))
    crearTaula('Tweet', cols)
    cols = []
    cols.append(ColumnDescriptor( name='tweet_id' ))
    crearTaula('Userline', cols)
    cols = []
    cols.append(ColumnDescriptor( name='tweet_id' ))
    crearTaula('Timeline', cols)
    print client.getTableNames()
```

Listing 60: *Script* per a crear les taules a HBase

Al principi i al final hi ha dos línies que mostren per pantalla les taules existents a la base de dades, així podem comprovar si al finalitzar l'execució de l'*script* les taules han estat creades satisfactòriament, un exemple de la sortida es pot veure a la figura **??**.

```
[]
['Timeline', 'Tweet', 'User', 'Userline']
```

Listing 61: Sortida de l'script

5.2.3 queries.py

El fitxer que a Twissandra es diu cass.py, a Twitbase el vaig reanomenar queries.py. El contingut ve a ser pràcticament el mateix que el de Twissandra, però adaptat per a usar HBase. El més problemàtic i que em va portar més

temps és la manera de funcionar HBase amb Thrift, el que amb Pycassa eren simples crides molt elementals, amb Thrift passen a ser més complicades, i al principi em va costar bastant treballar-hi.

Poc a poc vaig anar agafant experiència i al final em vaig construir unes funcions auxiliars que em van facilitar molt la feina.

Objectes i mutacions

La gran diferència al passar a usar Thrift amb HBase, és la manera de tractar els objectes. Twissandra, treballa tota l'estona amb objectes amb notació JSON, mentre que a Twitbase tot han de ser mutacions de columnes. Una mutació de columna és com s'anomena amb Hbase un *update*. I HBase funciona a base d'*updates*. Encara que vulguem inserir dades, l'operació per a fer-ho és un *update*, i HBase si no troba la fila que volem actualitzar la crea. Però tot funciona a base d'*updates*, inclús l'operació per eliminar una fila és un *update* amb un flag especial que marca les dades per a ser eliminades.

A Twissandra el contingut de la variable que conté un *Tweet*, abans de desar-lo és:

```
{
    'body': 'Tweet de prova',
    'user_id': '26a4078e-8449-11df-8bd6-0800276c99ff',
    'id': '5b61595e-1274-11e0-b9a7-080027ec4364'
}
```

Listing 62: Estructura d'un Tweet a Twissandra

```
[
    Mutation(column='body', isDelete=False, value='Tweet de prova'),
    Mutation(column='user_id', isDelete=False, value='emili'),
    Mutation(
        column='id',
        isDelete=False,
        value='85cd3a62-1276-11e0-b159-080027ec4364'
    )
]
```

Listing 63: Estructura d'un Tweet a Twitbase

Com podem veure a les figures 62 i 63, les estructures de dades son molt diferents, i mentre que les estructures de dades de Twissandra son fàcils de

tractar, les de Twtitbase son bastant més complicades de tractar i generar. Tota l'aplicació de Twissandra està preparada per a treballar amb els objectes amb una estructura com la de la figura 62, per tant el que vaig acabar fent va ser dues funcions auxiliars, una que convertís objectes tipus Twissandra a objectes tipus Twitbase i una que agafés una fila tal com es llegeix d'Hbase i la passés a un objecte del tipus que es fa servir a Twissandra.

```
def object_to_mutations(obj):
    """
    Transforms an object into a series of mutations
    """
    muts = []
    for attr in obj:
        mut = Mutation(column=attr, value=str(obj[attr]))
        muts.append(mut)
    return muts
```

Listing 64: Funció per a passar objectes a conjunts de mutacions

Amb aquesta funció, fem la transformació entre un objecte com el de la figura 62 i el passem a un conjunt de mutacions com es pot veure a la fugira 63.

```
def row_to_object(row):
    """
    Transforms an HBase row into an object
    """
    obj = dict()
    for attr in row.columns:
        aux = attr.split(":")
        if aux[1] != '' :
            obj[aux[1]] = row.columns[attr].value
        else:
            obj[aux[0]] = row.columns[attr].value
    return obj
```

Listing 65: Funció per a convertir una fila d'HBase a un objecte tipus Twissandra

Amb aquesta altra funció, el que fem és passar un objecte tal com es llegeix d'HBase com es mostra a la figura , i el convertim a un objecte com el que es mostra a la figura 62.

```
TRowResult
    (columns={
        'user_id:' : TCell(timestamp=1293536108147L, value='emili'),
        'body:' : TCell(
            timestamp=1293536108147L,
            value='Tweet de prova'
        ),
        'id:' : TCell(
            timestamp=1293536108147L,
            value='85cd3a62-1276-11e0-b159-080027ec4364'
        )
    },
    row = '85cd3a62-1276-11e0-b159-080027ec4364'
    )
```

Listing 66: Contingut d'una fila d'HBase tal com arriba a Twitbase

Cada vegada que volem desar un objecte a HBase, hem de fer una crida a la funció de la figura 64, i cada vegada que llegim una fila d'HBase l'hem de passar per la funció 66. D'aquesta manera, evitem qualsevol problema amb la resta de l'aplicació ja que els objectes passen a tenir la forma que tenien amb Twissandra.

Funcions

Una vegada solucionada la transformació dels objectes entre HBase i Twissandra, ja només em calia "traduir"totes les funcions i fer-les funcionar sobre HBase. En aquest apartat mostro algunes funcions per a reflectir el funcionament d'HBase amb Thrift.

```
def get_tweet(tweet_id):
    """
    Given a tweet id, this gets the entire tweet record.
    """
    try:
    tweet = client.getRow('Tweet',str(tweet_id))
    except NotFoundException:
        raise NotFound('Tweet %s not found' % (tweet_id,))
    return row_to_object(tweet[0])
```

Listing 67: Funció que retorna un Tweet de la base de dades a partir del seu id

A la figura 67 veiem una funció que simplement agafa un Tweet d'HBase a partir del seu id. El codi és absolutament trivial, només fa servir una crida de Thrift anomeda *getRow*. Per acabar la funció, al retornar l'objecte del Tweet, es passa el resultat obtingut d'HBase per la funció que hem vist abans, *row_to_object*.

A part d'aquest tipus de funcions consultores que son trivials, en podem trobar algunes altres de més complexes.

```
1  def add_friends(from_user, to_users):
2      """
3      Adds a friendship relationship from one user to some others.
4      """
5      muts = []
6      for user_name in to_users:
7          mut = Mutation(column="friends:"+str(user_name),
8                  value=str(user_name))
9          muts.append(mut)
10         mut = Mutation(column="followers:"+str(from_user),
11                 value=str(from_user))
12         client.mutateRow("User",str(user_name), [mut])
13     client.mutateRow("User", str(from_user), muts)
```

Listing 68: Funció que crea relacions d'amistat entre usuaris

La funció de la figura 68 és la que crea les relacions d'amistat entre els usuaris. Té com a paràmetres d'entrada el nom d'un usuari (*from_user*) i una llista d'usuaris (*to_users*). El que fa la funció és afegir a la fila de l'usuari *from_user*, a la columna friends, una entrada per cada usuari de la llista *to_users*. I a cada usuari de la llista *to_users*, afegeix una entrada a la columna followers amb referència a l'usuari *from_user*.

Com es pot veure la complexitat es troba en aconseguir abstreure els canvis necessaris a mutacions. Una vegada es tenen les mutacions, les operacions son trivials. Per últim veiem com és la funció que elimina les relacions d'amistat entre usuaris.

```
def remove_friends(from_user,to_users):
    """
    Removes a friendship relationship from one user to some others.
    """
    for user_name in to_users:
        mut = Mutation(column='friends:'+user_name, isDelete='true')
        client.mutateRow('User', str(from_user), [mut])
    for to_user_name in to_users:
        mut = Mutation(column='followers:'+from_user, isDelete='true')
        client.mutateRow('User', str(to_user_name), [mut])
```

Listing 69: Funció que elimina relacions d'amistat entre usuaris

Com es pot veure a la figura 69, el sistema és exactament el mateix. S'han de crear les mutacions pertinents i executar-les. La peculiaritat d'aquesta funció és que eliminar una dada no té una funció per a liminar com es podria esperar, sinó que el que fem és una mutació (*update*) i marcar la dada amb el *flag isDelete* que com em vist al capítol d'HBase, eliminarà la dada quan es faci una compactació.

5.2.4 Divergències entre HBase i Cassandra

Com hem vist als capítols sobre HBase i Cassandra, existeixen moltes diferències entre aquests dos sistemes. Algunes d'aquestes es veuen clarament al tractar amb aquestes dues aplicacions. La diferència que més influeix a l'hora de desenvolupar aplicacions, és que Cassandra és una base de dades més enfocada a desenvolupar aplicacions, mentre que HBase està més enfocada a l'anàlisi de dades. Això ho veiem clarament a la funció que retorna els Tweets de la base de dades.

```
    tweets = TWEET.multiget(timeline.values())
    tweets = dict(((t['id'], t) for t in tweets.values()))
    ordered = [tweets.get(tweet_id) for tweet_id in timeline.values()]
```

Listing 70: Fragment de la funció per a obtenir els Tweets de la bd amb Twissandra

A la figura 70 podem veure com en 3 línies obtenim tots els tweets que volem mostrar, i després els ordenem gracies a la ordenació que ens ofereix Cassandra. La ordenació es fa mapejant els valors a una estructura orderedDict com hem vist a la figura 56.

```
1       tweets = []
2       timeline = client.getRow(str(cf), str(user_name))
3       timeline = timeline[0]
4       for val in timeline.columns:
5           tweet_id = timeline.columns[val].value
6           tweet_row = client.getRow("Tweet", str(tweet_id))
7           tweet_row = tweet_row[0]
8           tweet = row_to_object(tweet_row)
9           tweets.append(tweet)
```

Listing 71: Fragment de la funció per a obtenir els Tweets de la bd amb Twitbase

Per a fer el mateix que a la figura 70 a Twitbase hem de fer el que es mostra a la figura 71. I al codi existent encara hi falta afegir l'ordenació. Per tant, tot aquest codi equival únicament a la línia 1 de la figura 70.

5.2.5 Limitacions de Twitbase

Twitbase té exactament les mateixes funcionalitats que té Twissandra. Totes les funcions que s'han presentat al capítol de Twissandra estan implementades a Twitbase i funcionen bé. Per tant a nivell de funcionalitats de la web, tenen exactament les mateixes.

La diferència més la trobem en el tractament de les sessions d'usuari. A Twissandra, quan usuari es registra, automàticament passa a tenir una sessió registrada i pot fer servir totes les funcionalitats de la web. A Twitbase en canvi, després de registrar-se ha de fer el login. Les sessions d'usuari de Twitbase tenen una duració molt limitada, ja que a la mínima que sorgeix un error, o al cap de poc temps d'estar loguejat, el sistema automàticament perd la sessió de l'usuari i aquest passa a estar com no identificat.

Aquests problemes tenen més a veure amb el funcionament de Django que amb la capa de base de dades, per tant queden com a millores a fer al programa. Simplement he decidit no abordar aquestes millores durant aquest projecte ja que queden fora del seu abast.

5.3 conclusions

Sempre que es comença a treballar amb una tecnologia que es desconeix, és necessari un procés llarg d'aprenentatge. La gran part d'aquest procés consisteix en llegir documentació i entendre a fons aquesta tecnologia. Però

només amb llegir i entendre no n'hi ha prou. Sempre passa que llegim documents i ens sembla que dominem una matèria fins que comencem amb la pràctica i ens adonem que ens falta molt per aprendre. Això és exactament el que em va passar amb aquest cas pràctic. Després d'un temps llegint tota la documentació que vaig trobar per Internet vaig decidir llançar-me a fer aquest cas pràctic.

A mida que provava de fer coses em veia cada vegada més perdut i no veia gens clar com abordar Twitbase. Per tant, quan ja tenia alguna petita cosa funcionant, vaig decidir tornar a començar pel principi i llegir més documentació. No se si va ser que abans no havia sabut buscar prou bé o que en l'interval de temps que vaig estar fent proves van aparèixer molts documents (tot i que segurament va ser el fet de començar de nou però amb una base de coneixement). Però el fet és que al tornar a buscar documentació vaig trobar molts millors articles i exemples que em van ajudar a entendre molt millor el món NoSQL.

Per tant aquest cas pràctic va complir amb escreix el seu objectiu, que era aconseguir la familiaritat amb els sistemes NoSQL. Una vegada vaig estar millor documentat, el procés va ser bastant més agradable. Tot i que vaig tenir bastants problemes fins que vaig agafar pràctica amb el pas d'objectes a mutacions, al final vaig aconseguir tenir una versió de Twitbase totalment funcional.

Tot aquest procés d'aprenentatge barrejant documentació i pràctica, em va fer veure que el camí que havia triat no era el més indicat. Després de llegir molt sobre els dos sistemes de bases de dades i de practicar amb ells, vaig veure que els dos estaven enfocats a diferents camps i que jo havia provat de fer una mateixa aplicació amb els dos sistemes. Mentre que Cassandra està molt més enfocat al desenvolupament d'aplicacions, HBase està molt més enfocat a l'anàlisis de dades. Cassandra ofereix moltes més operacions per facilitar la feina al desenvolupador d'aplicacions (com el multiget i l'ordenació mostrada dos capítols enrere), mentre que HBase el que busca és facilitar l'anàlisi de dades mitjançant lectures ràpides i integració amb Hadoop.

Així que una vegada vaig tenir acabat Twitbase, vaig decidir que no era l'aplicació adient per a seguir al segon cas pràctic. Que el que havia de buscar era crear dos tipus d'entorns diferents, un per a Cassandra i un per HBase. Per a Cassandra volia un entorn dinàmic, fer-la servir com a base de dades del model d'una aplicació web. Mentre que per HBase la idea era muntar un sistema que desés logs sobre HBase per a poder-ne fer anàlisis amb Hadoop usant eines com Hive o Pig.

Per tant, la realització d'aquest cas pràctic em va servir per agafar la suficient confiança amb els entorns NoSQL com per a plantejar un nou cas pràctic i deixar de banda Twitbase.

Capítol 6

Segon cas pràctic

Amb l'experiència del primer cas pràctic, havent treballat sobre Cassandra i HBase i amb la suficient base teòrica, vaig plantejar un segon cas pràctic. L'objectiu d'aquest cas pràctic era treballar amb les dues bases de dades però donant a cada un us diferent. Treballar amb Cassandra com a base de dades d'una aplicació, i HBase com a magatzem de logs sobre el que poder fer anàlisi de dades a posteriori. Amb aquests dos enfocs diferents per a cada base de dades, l'objectiu final era analitzar les dues eines i veure el seu rendiment en els entorns en que teoria treballaven millor cadascuna.

Com que tot el primer cas pràctic es va realitzar sobre Twissandra, i ja hi estava familiaritzat, vaig decidir seguir treballant amb aquesta aplicació per aquest segon cas pràctic. Així vaig separar aquest segon cas pràctic en dues parts:

1. Generar un banc de proves per a Twissandra per a evaluar el rendiment de Cassandra

2. Instrumentar Twissandra per a generar logs sobre HBase i fer anàlisi d'aquests logs

Per a la primera part es tractava de construir un banc de proves fidel a la realitat. Un test de proves que permetés provar Twissandra com si es trobés en producció obert al públic. Per a fer-ho res millor que agafar les dades directament de Twitter.

Una vegada tingués el test de proves implementat, el que havia de fer era instrumentar Twissandra per a emmagatzemar logs sobre HBase. Una vegada tingués aquests logs, la intenció era usar les eines adients per analitzar grans quantitats de logs, com per exemple Hadoop a través de Pig.

6.1 Obtenció dels models de comportament de Twitter

Buscant un entorn el més real possible, vaig decidir obtenir models de comportament de Twitter, ja que Twissandra és una emulació de Twitter, què millor que fer-hi proves amb sets de dades obtinguts de l'original.

Com el que volia era emular l'entorn real de Twitter, em calien les dades que em permetessin fer-ho. Em calia poder emular per a qualsevol moment del dia la càrrega real del sistema. Per a fer-ho em calien les següents dades:

- Tweets per segon, en funció de l'hora del dia
- Models de comportament dels usuaris

Amb aquesta informació podria simular amb molta fidelitat la càrrega del sistema, ja que per a cada moment del dia podria simular la càrrega real, i tenint els models d'ús de diferents usuaris, podria generar la càrrega necessària segons aquests models. Això em permetia no només fer els test de càrrega el més real possible, sinó poder re-dimensionar aquesta càrrega realísticament. Podent generar la càrrega per a qualsevol moment de temps, podia també simular que passaria per exemple si l'ús s'incrementés en un x per cent, i veure com reaccionaria el sistema en general i Cassandra en particular a aquests canvis.

6.1.1 API de Twitter

Per a obtenir aquests models de comportament, em vaig documentar i vaig veure que Twitter oferia una API[1] molt complerta. La API de Twitter està formada per dos grans parts:

1. API de consultes
2. API de *streaming*

La primera API és una API basada en REST, que ens permet realitzar qualsevol tipus de consulta sobre el sistema i aquest ens retorna la informació desitjada. Per exemple ens permet realitzar una cerca dels últims Tweets d'un cert usuari, o els últims Tweets que continguin un determinat conjunt de paraules. Aquesta API està dividida en dues parts, una és únicament per a fer cerques, però per a aquest cas pràctic no la farem servir. Les API's REST de Twitter permeten obtenir els resultats en format JSON i XML.

[1] Application programming interface

La segona API, és una API menys convencional. La API d'*streaming* ens permet obrir un canal de comunicació amb Twitter i obtenir resultats en temps real. Per exemple, en comptes de fer una cerca i obtenir in nombre limitat de resultats, com amb l'altre API, amb aquesta API podem fer una cerca i aleshores anem rebent en temps real els Tweets que rep Twitter i contenen els mots que hem indicat. Aquesta API només permet rebre resultats en format JSON.

Autenticació

Les API's de Twitter tenen certes crides que requereixen autenticació. Per a autentificar les peticions fan servir l'estàndard OAtuth, i per a obtenir una autenticació OAuth, cal demanar un id de desenvolupador. Per tant vaig haver de registrar una aplicació per a que em donessin permís per accedir a algunes de les funcions de la API que necessitaven autenticació.

Restriccions

Twitter imposa un model de restriccions sobre les seves API's. La gran majoria de crides tenen una limitació d'ús horària. Aquesta limitació canvia depenent de la consulta, però en general, les crides estan limitades a 350 crides per usuari i hora.

Això em va obligar a pensar bé el codi i mirar d'aprofitar al màxim les crides, ja que volia extreure el màxim d'informació però tenia temps limitat.

6.1.2 Eines usades

Per a la gran majoria d'aquest cas pràctic vaig desenvolupar les diferents eines que em van caldre en Java, ja que és el llenguatge que més faig servir. La majoria de les aplicacions les vaig programar amb un entorn de servlets i jsp's, ja que és un entorn molt còmode per a treballar i que m'oferia un entorn gràfic més agradable de fer servir que una aplicació per la consola.

Per a facilitar-me la feina en varis aspectes de l'aplicació vaig usar dues llibreries, Twitter4j i Hector.

Twitter4j

Per a mirar de simplificar l'ús de les API's i minimitzar el cost de desenvolupament vaig decidir fer servir una llibreria anomenada Twitter4j.

Aquesta llibreria oferia un client en Java que disposava de la gran majoria de crides de la API i oferia mecanismes de control d'errors, autentificació amb OAuth, etc. Que feia que se'm simplifiqués la feina.

A la pràctica aquesta llibreria tot i que em va ajudar en alguns aspectes, és una aplicació molt mal documentada. Tota la web té varis tutorials d'ajuda i fragments de codi d'exemple, però aquests son per a versions antigues de la llibreria que no funcionen amb les versions actuals. Així que el que havia de ser una tasca trivial va acabar sent bastant més complicat.

El gran problema que vaig trobar amb Twitter4j va ser a l'hora de fer l'autenticació amb OAuth. A part de la mala documentació de la llibreria, vaig coincidir amb una renovació de l'API de Twitter que va desfasar encara més tota la documentació existent. De fet, vaig haver de posposar l'ús de l'autenticació durant un temps fins que vaig trobar un article que em va ser de gran ajuda.

Algunes crides de l'API de Twitter requereixen que les consultes estiguin autentificades mitjançant OAuth. Per a fer-ho cal demanar un id de desenvoupador. Així que vaig registrar una aplicació anomenada pfcCrawler. Una vegada vaig tenir els codis d'autenticació, vaig desenvolupar una petita aplicació per a demanar el token d'autenticació i poder usar les consultes restringides.

Per a poder-se autentificar amb OAuth cal seguir els següents passos:

1. Registrar una aplicació per a obtenir un codi d'aplicació i una contrasenya

2. Demanar un authorization token per al client

3. El client que genera l'authorization token ha de confirmar que dona accés a l'aplicació

4. Una vegada confirmada l'autorització per part del client, Twitter confirma aquesta autorització donant un Access Token.

Aquest procés només cal fer-lo una vegada per usuari, en el meu cas, només una vegada en total ja que jo soc l'únic usuari d'aquesta aplicació.

Per a fer aquests passos vaig haver de fer dues classes en Java. La primera per a generar la petició d'authorization token per al client. Aquesta classe està formada per dues funcions, un main i una funció auxiliar.

```
1  public static void main(String[] args)throws TwitterException,IOException{
2      Twitter twitter = new TwitterFactory().getInstance();
3      twitter.setOAuthConsumer("kcddMU5GZcVuBaPNN6hJ5w",
4          "yD0ycyd4MCMUn2qAB4GPJq7J53D4L1CqQWarCs9Tc0");
5      RequestToken requestToken = twitter.getOAuthRequestToken();
6      AccessToken accessToken = null;
7      BufferedReader bufferedReader = new BufferedReader(
8          new InputStreamReader(System.in));
9      while (null == accessToken) {
10         System.out.println("Accedeix a la seguent url
11             i autoritza l'acces al teu compte:");
12         System.out.println(requestToken.getAuthorizationURL());
13         System.out.print("Introdueix el PIN obtingut:");
14         String pin = bufferedReader.readLine();
15         try {
16             if (pin.length() > 0) {
17                 accessToken = twitter.getOAuthAccessToken(requestToken,
18                     pin);
19             } else {
20                 accessToken = twitter.getOAuthAccessToken();
21             }
22         } catch (TwitterException e) {
23             if (401 == e.getStatusCode()) {
24                 System.out.println("No s'ha pogut
25                     obtenir el Token d'acces.");
26             } else {
27                 e.printStackTrace();
28             }
29         }
30     }
31     storeAccessToken(accessToken);
32     System.exit(0);
33 }
```

Listing 72: Generació l'accessToken per al client

El codi de la figura 72 mostra el codi necessari per a l'obtenció d'un accessToken per a un client. El primer que s'ha de fer és identificar l'aplicació que està demanant l'accés. Això es fa a les línies 2 i 3, posant les credencials que ens dona Twitter al registrar l'aplicació a la seva web.

El següent pas és re-dirigir a l'usuari a una url que és la que usa Twitter per a garantir que l'usuari dona accés a l'aplicació per a accedir al seu compte. Això és el que fa la línia 12.

Una vegada el client autoritza l'aplicació, Twitter li proporciona un PIN, aleshores l'usuari l'introdueix a l'aplicació mitjançant la consola i es prova

d'obtenir l'accessToken definitiu (linia 17).

Si tot funciona com és degut, al final es té l'accessToken corresponent i es desa a disc (lína 31).

```
1  private static void storeAccessToken(AccessToken accessToken) {
2      try {
3          FileOutputStream fileOutputStream = new FileOutputStream(
4              "token.txt");
5          ObjectOutputStream objectOutputStream = new ObjectOutputStream(
6              fileOutputStream);
7          objectOutputStream.writeObject(accessToken.getToken());
8          objectOutputStream.flush();
9          fileOutputStream = new FileOutputStream("tokenSecret.txt");
10         objectOutputStream = new ObjectOutputStream(fileOutputStream);
11         objectOutputStream.writeObject(accessToken.getTokenSecret());
12         objectOutputStream.flush();
13     } catch (IOException e) {
14         e.printStackTrace();
15     }
16 }
```

Listing 73: Persistència de l'accesToken

A la figura 73 es mostra el codi per desar l'accessToken al disc. Això és molt útil, ja que una vegada l'usuari ha donat permís a l'aplicació, ens podem guardar l'access token i ja no cal que tornem a fer aquest procés.

Per últim vaig fer una altra classe que servia per a recuperar l'accessToken del disc i poder fer peticions a les crides restringides. Aquest procés és molt més senzill i és el codi que es farà servir a la resta de classes que es comuniquin amb Twitter.

```
1 TwitterFactory factory = new TwitterFactory();
2 AccessToken accessToken = loadAccessToken();
3 Twitter twitter = factory.getOAuthAuthorizedInstance(
4     "kcddMU5GZcVuBaPNN6hJ5w",
5     "yD0ycyd4MCMUn2qAB4GPJq7J53D4L1CqQWarCs9Tc0",
6     accessToken);
```

Listing 74: Obtenció d'una instància de Twitter autentificada

Amb aquestes 4 línies de codi és com obtindrem les instàncies de Twitter4j autentificades amb OAuth.

```
1  private static AccessToken loadAccessToken() {
2      String token = null;
3      String tokenSecret = null;
4      try {
5          FileInputStream fileInputStream = new FileInputStream("token.txt");
6          ObjectInputStream objectInputStream = new ObjectInputStream(
7              fileInputStream);
8          token = (String) objectInputStream.readObject();
9          fileInputStream = new FileInputStream("tokenSecret.txt");
10         objectInputStream = new ObjectInputStream(fileInputStream);
11         tokenSecret = (String) objectInputStream.readObject();
12     } catch (IOException e) {
13         e.printStackTrace();
14     } catch (ClassNotFoundException e) {
15         e.printStackTrace();
16     }
17     return new AccessToken(token, tokenSecret);
18 }
```

Listing 75: Funció per a llegir l'accessToken desat al disc

La funció de la figura 75 serveix per a recuperar l'accessToken que s'ha guardat al disc amb la funció de la figura 73 .

Hector

Hector és una llibreria per a gestionar la connexió amb Cassandra. Totes les dades generades amb l'API de Twitter les vaig desar sobre Cassandra, i vaig usar aquesta llibreria per gestionar la connexió. El seu ús és molt senzill i no m'ha portat cap tipus de problema. A tots els fitxers on deso dades sobre Cassandra sempre tindré el mateix codi per a fer la connexió:

```
1 Cluster c = HFactory.getOrCreateCluster("pfcCluster", "localhost:9160");
2 Keyspace keyspace = HFactory.createKeyspace("TwitterCrawler", c);
```

Listing 76: connexió a Cassandra amb Hector

Amb les dues línies de codi mostrades a la figura 76, ja tenim la connexió establerta amb el clúster de Cassandra que es troba a localhost al port 9160[2] i al Keypsace anomenat *TwitterCrawler*.

[2]port per defecte de Thrift

6.1.3 Tweets per segon

Per a obtenir els Tweets per segon, vaig usar una crida de l'*Streaming API* anomenada sample. Aquesta crida el que fa és retornar l'1% del tràfic real de Twitter. Per tant, dona una mesura exacte del tràfic real, escalat cap a baix. Però pel que necessitava era perfecte.

Implementació

El que vaig fer va ser implementar un *Thread* que gestionés la crida a la API i tot el necessari per agafar les dades necessàries. Aleshores amb un *Servlet*, feia les crides necessàries per engegar i parar el *Thread*, així com per consultar-ne l'estat.

El primer que feia el Thread era connectar-se amb l'*Streaming API*.

```
1  @Override
2  public void run() {
3      tweetsRecieved = new Long(0);
4      lastRatioTweets = tweetsRecieved;
5      StatusListener listener = new StatusListener() {
6          @Override
7          public void onStatus(Status status) {
8              saveTweet(status);
9              tweetsRecieved++;
10         }
11         @Override
12         public void onDeletionNotice(
13             StatusDeletionNotice statusDeletionNotice) {}
14         @Override
15         public void onTrackLimitationNotice(
16             int numberOfLimitedStatuses) {}
17         @Override
18         public void onException(Exception ex) {}
19     };
20     twitterStream = new TwitterStreamFactory(listener).
21         getInstance("user", "pass");
22     twitterStream.sample();
23     scheduleRatio();
24 }
```

Listing 77: Funció run del *Thread*

Com es pot veure a la figura 77, primer s'inicialitzen un parell de comptadors que es faran servir més endavant, aleshores a la línia 5 s'inicialitza una

variable del tipus *StatusListener* que és el que ens cal per poder cridar a la funció *sample* a la línia 22. Per a crear la variable del tipus *StatusListener* hem d'implementar els diferents mètodes que volem executar depenent de l'event que rebem des de Twitter. Jo només vaig implementar *onStatus* que és el mètode que ens indica que hem rebut un Tweet.

Al rebre un Tweet el que fa l'aplicació és desar-lo a la base de dades i incrementar un comptador. Els Tweets es guardaven per si em feien algun servei al futur.

Per últim a la línia 23 es crida al mètode *scheduleRatio()*.

```
1  private void scheduleRatio(){
2      new java.util.Timer().schedule(
3          new java.util.TimerTask() {
4              @Override
5              public void run() {
6                  saveRatioByTime();
7              }
8          },
9           ratioTime
10     );
11 }
```

Listing 78: Funció *scheduleRatio*

La funció *scheduleRatio* l'únic que fa és iniciar un comptador que s'executa al cap d'un cert nombre de mili-segons indicat per la variable *ratioTime*. Aquesta variable està inicialitzada al principi del fitxer a 180000 milisegons, l'equivalent a 3 minuts. Per tant les mesures es prenen cada 3 minuts. La funció que s'executa cada 3 minuts s'anomena *saveRatioByTime*.

A la figura 79 veiem la funció *saveRatioByTime*. Aquesta funció s'executa regularment cada 3 minuts, i a cada execució calcula els Tweets per segon que s'han fet des de l'última execució de la funció (3 minuts abans), i guarda el resultat a Cassandra. Com veiem gran part de la funció serveix únicament per a formatar els resultats adequadament per a desar-los a Cassandra.

El codi mostrat és l'essencial del *Thread* per a obtenir la relació de Tweets per segon de Cassandra. Per a executar aquest codi i mostrar-ne l'estat, vaig implementar un petit *Servlet* i un jsp.

```
private void saveRatioByTime(){
    StringSerializer se = new StringSerializer();
    Mutator<String> m = HFactory.createMutator(keyspace,  se);
    float tweets = tweetsRecieved - lastRatioTweets;
    float ratio = tweets / (float)(ratioTime/1000);
    lastRatioTweets = tweetsRecieved;
    Format decimal = new DecimalFormat( "#######00.00");
    Format dayFormat = new SimpleDateFormat("ddMMyyyy");
    Format timeFormat = new SimpleDateFormat("HHmmss");
    Date now = new Date();
    m.insert(dayFormat.format(now), "TweetStreamingRatioByTime",
        HFactory.createStringColumn(timeFormat.format(now),
        decimal.format(ratio)));
    scheduleRatio();
}
```

Listing 79: Funció *saveRatioByTime*

```
if(op.equals("start")){
    crawler.run();
    result.put("status", "running");
}else if(op.equals("stop")){
    crawler.stop();
    result.put("status", "stopped");
}else if(op.equals("status")){
    if(crawler.isRunning()){
        result.put("status", "running");
        long numTweets = crawler.getTweetsRecived();
        result.put("numTweets", numTweets);
        Long start = crawler.getStartTime().getTime();
        Long now = new Date().getTime();
        Long milisRunning = now - start;
        result.put("milisRunning", milisRunning);
    }else{
        result.put("status", "stopped");
    }
}
```

Listing 80: *Servlet*

A la figura 80 es mostra el cos del *Servlet* que interactua amb el *Thread* que obté les dades de Twitter. Com es pot veure ofereix 3 funcions per a ser executades des d'el jsp:

1. Start

2. Stop

3. Status

La única que potser no és obvia és la funció d'*status*, que serveix per a comprovar si el *Thread* està funcionant. En cas positiu retorna un objecte JSON amb el nombre de Tweets que s'han obtingut des de que el *Thread* ha entrat en funcionament, i el temps que fa que està funcionant.

El jsp mostra les dades tal com es mostra a la següent figura:

Listing 81: *jsp* que mostra l'estat del *Thread* de consulta

Resultats

Amb la implementació que acabem de veure, ja podia recollir tota la informació que em calia sobre la quantitat de Tweets per segon que rep Twitter. Cada 3 minuts prenia una mesura i la desava a Cassandra. Els resultats els guardava en una única família de columnes amb la següent forma:

```
dia : {
    hora: {
        ratio
    }
}
```

Listing 82: Esquema de dades a Cassandra

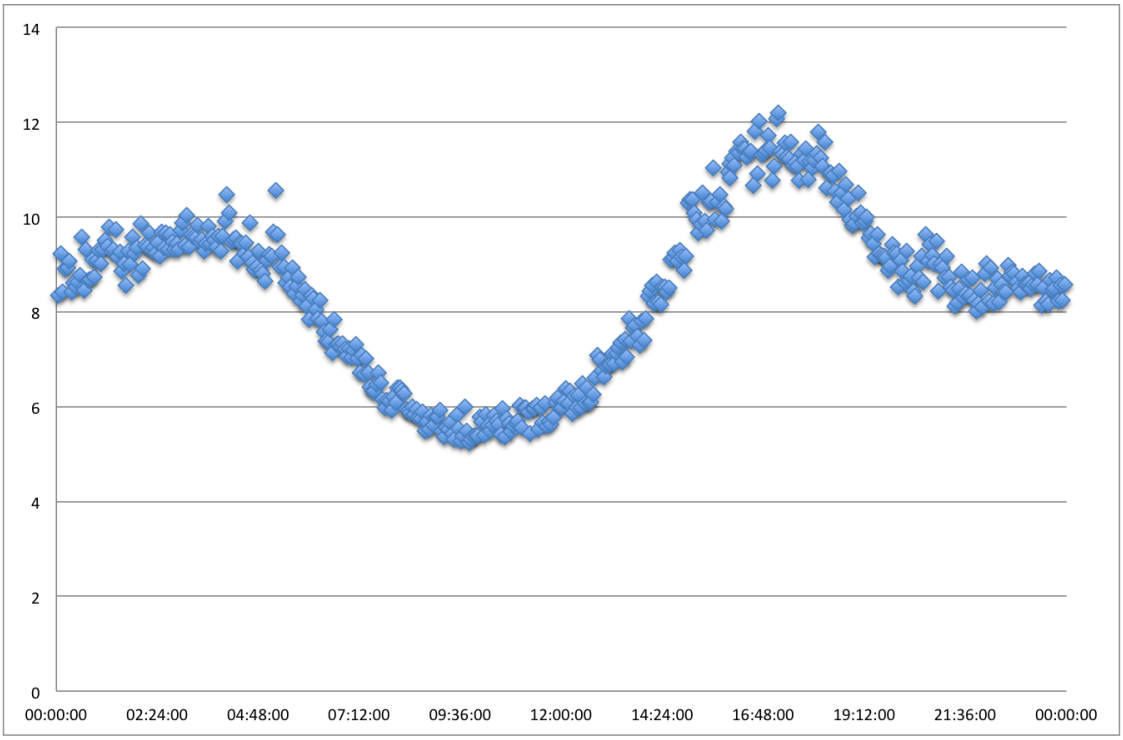
L'esquema de dades constava d'una família de columnes on cada fila corresponia a un dia, i cada fila tenia una columna per a cada mesura. Les claus de les columnes son les hores de mesura i els seus valors el ratio Tweets / segon per aquella hora.

```
22102010 : {
    (column=100020, value=05,98, timestamp=1287734420131000)
    (column=100320, value=06,00, timestamp=1287734600132000)
    (column=100620, value=06,29, timestamp=1287734780138000)
    (column=100920, value=05,68, timestamp=1287734960140000)
}
```

Listing 83: Mesures del dia 22 d'Octubre a les 10:00

A la figura 83 veiem un exemple de mesures, en concret per al dia 22 d'Octubre del 2010, i entre les 10:00 i les 10:09 del matí.

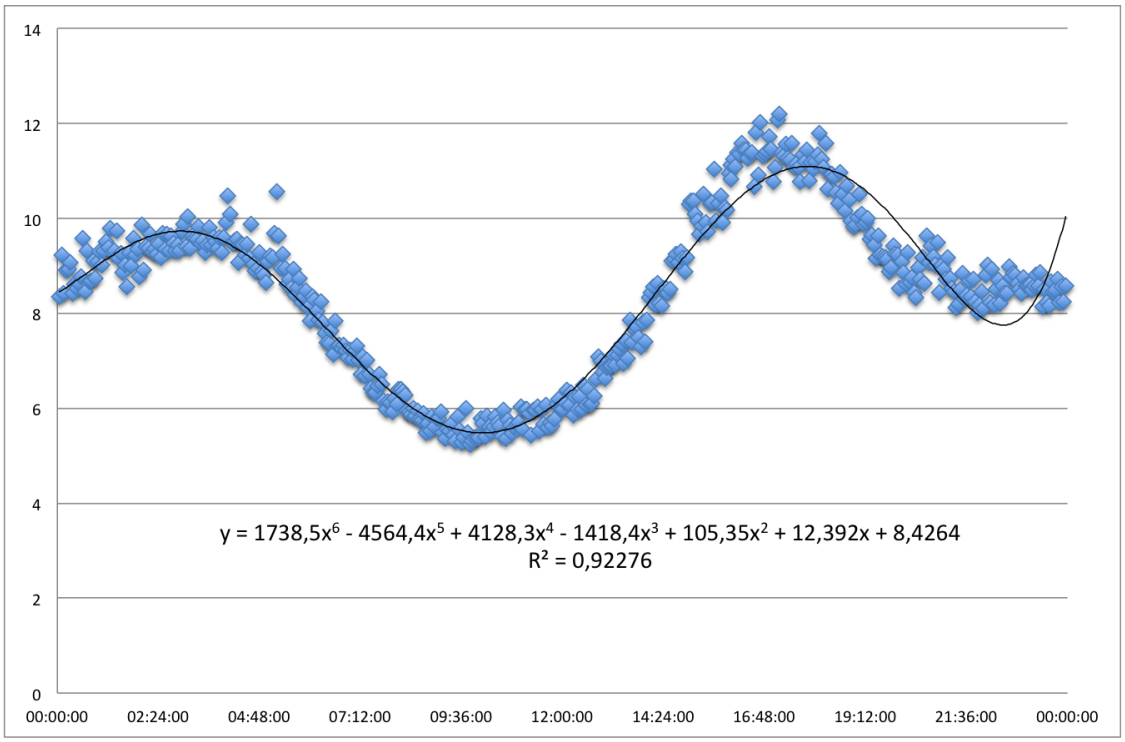
Obtenir les mesures per a un dia sencer no era trivial, ja que s'havia d'executar el programa durant 24 hores sense interrupció, i va resultar que la implementació de la llibreria Twitter4j tenia alguns problemes i sempre es desconnectava passades unes hores. Però al final tot va funcionar correctament i vaig obtenir mesures per a un dia sencer.



Listing 84: Observacions per a un dia sencer

Com es pot observar a la figura 84, el ratio de Tweets/segon és bastant

fluctuant i oscil·la entre els $5,2$ i els $12,2$ Tweets/segon. L'hora amb més activitat es troba entre les 16:00 i les 18:00. Si busquem una explicació al perquè aquesta és l'hora amb més activitat, segurament la resposta és que son les hores que coincideixen amb el matí als EEUU, mentre que les hores de menys activitat, entre les 8:00 i les 12:00 del nostre matí coincideixen amb la seva matinada.



Listing 85: Línia de tendència

A la figura 85, veiem el resultat de sobreposar a la gràfica una línia de tendència polinòmica de grau 6. El valor de l'equació d'aquesta línia és

$$y = 1738,5x^6 - 4564,4x^5 + 4128,3x^4 - 1418,4x^3 + 105,35x^2 + 12,392x + 8,4264$$

mentre que el seu R^2 és $R^2 = 0,92276$.

Amb aquestes dades ja en tenim suficient per al que volem fer. Per tant ara ja només ens cal obtenir models d'ús per a diferents usuaris.

c

6.1.4 Models d'usuari

Una vegada ja podem simular la càrrega per segon sobre Twissandra, ens calen diferents models d'usuari per a generar una càrrega realista. Per agafar usuaris, el primer que vaig fer va restringir el rang de cerca a usuaris que portessin més de 3 mesos registrats a Twitter, ja que així les estadístiques que recollís serien més complertes. Per raons de rendiment, com veurem més endavant, també vaig limitar la cerca a usuaris amb menys d'un milió de seguidors.

Les dades que volia recollir referents als usuaris eren:

- Interval de temps entre Tweets
- Llargada mitja dels Tweets
- Nombre de seguidors

Als pròxims apartats veurem el perquè d'aquestes dades i el codi necessari per a obtenir-les.

Interval de temps entre Tweets

L'interval de temps entre Tweets és essencial, ja que a l'hora de simular la càrrega del sistema, el que farem serà agafar l'usuari que per estadística ha de fer el pròxim Tweet. Si per exemple tenim un usuari que fa un Tweet cada 10 minuts a la vida real i un que en fa un cada hora, al nostre entorn de simulació sabem que un n'haurà de fer 6 vegades més que l'altre. Per tant aquest és l'indicador de més valor que hem de recollir.

Llargada mitja dels Tweets

Per a fer una simulació el més realista possible, necessitem saber el tipus de Tweet que fa cada usuari. Si un usuari sempre escriu Tweets per sota dels 40 caràcters i un altre en canvi, sempre aprofita els 140 caràcters, hem de reflectir-ho a l'hora de fer la simulació. Per a fer aquest projecte, únicament vaig fer la mitja de la llargada dels últims 100 Tweets de l'usuari. Tot i que per fer-ho més realista ens caldria guardar aquesta mitja i la desviació estàndard per tal de poder recrear amb més exactitud els Tweets de cada usuari. Per a no complicar massa el sistema, per aquest projecte només farem servir la mitja lineal.

Nombre de seguidors

Si agaféssim només els dos paràmetres que acabem d'explicar, ja en tindríem prou per a fer una simulació del comportament dels clients realista . Però el que nosaltres volem avaluar realment, és el comportament del servidor. Observem com és la inserció d'un Tweet a Twissandra:

```
1  def save_tweet(tweet_id, username, tweet):
2      ts = long(time.time() * 1e6)
3      tweet['body'] = tweet['body'].encode('utf-8')
4      TWEET.insert(str(tweet_id), tweet)
5      USERLINE.insert(str(username), {ts: str(tweet_id)})
6      USERLINE.insert(PUBLIC_USERLINE_KEY, {ts: str(tweet_id)})
7      follower_usernames = [username] + get_follower_usernames(username)
8      for follower_username in follower_usernames:
9          TIMELINE.insert(str(follower_username), {ts: str(tweet_id)})
```

Listing 86: Codi de Twissandra per a desar un Tweet

A la figura 6.4.1 es pot veure el codi que empra Twissandra per a desar els Tweets. El primer que fa és obtenir el *timestamp* del moment de la inserció, després es codifica el cos del missatge per a assegurar que es podrà llegir correctament. Aleshores, entre la línia 4 i 9 es fan una sèrie d'insercions. Les 3 primeres son, primer desar el Tweet, després posar-ne una referència al *Userline* de l'usuari que fa el Tweet, i per últim s'en desa una referència al *Userline* públic. Una vegada fet això ve el pas clau (línies 7-9), s'obtenen els noms de tots els seguidors de l'usuari, i es desa una referència al Tweet al *Timeline* de cadascun dels seguidors.

En resum, per cada Tweet que es desa al sistema cal fer $3 + n$ insercions, on n és el nombre de seguidors de l'usuari. Així el cost al servidor per cada Tweet va directament relacionat al nombre de seguidors dels usuaris.

Obtenció de les mètriques d'usuari

Ara que ja hem vist quines dades ens calia obtenir per a cada usuari, passem a veure el procés i el codi necessari per a obtenir-les.

El primer i més important de tot, era trobar usuaris de Twitter. Com s'ha explicat abans, al obtenir la mètrica de Tweets per segon, també es desaven tots els Tweets a una família de columnes de Cassandra. Per tant, ja tenia la font d'on obtenir noms d'usuari. El que vaig fer va ser fer consultes amb

clau aleatòria sobre la taula de Tweets [3]. Amb cada consulta obtenia 100 Tweets aleatoris.

```
1  RangeSlicesQuery sliceQuery = HFactory.
2  createRangeSlicesQuery(keyspace, se, se, se);
3  Date now = new Date();
4  sliceQuery.setKeys(Long.toString(now.getTime()), "");
5  sliceQuery.setRange("", "", false, 100);
6  sliceQuery.setColumnFamily("Tweet");
7  sliceQuery.setColumnNames("fromUser:ScreenName");
8  try{
9      QueryResult<Rows<String, String, String>> tweets =
10     sliceQuery.execute();
11     String[] userNames = new String[100];
12     int i = 0;
13     for (Row<String, String, String> row : tweets.get()) {
14         ColumnSlice<String, String> slice = row.getColumnSlice();
15         for (HColumn<String, String> column : slice.getColumns()) {
16             userNames[i] = column.getValue();
17             i++;
18         }
19     }
20 }catch(Exception ex){}
```

Listing 87: Codi per a obtenir els usuaris de Cassandra

A la figura 87 tenim el codi necessari per a obtenir 100 Tweets aleatoris de Cassandra. Per a fer-ho creem una *SliceQuery*, que el que fa es retornar-nos un fragment de la taula, que limitem a 100 valors. Una vegada obtinguts els usuaris, iterem sobre el resultat i guardem els seus noms d'usuari a un *Array* anomenat *userNames*.

Entre les línies 19 i 20 de la figura 87, ve aleshores el codi de la figura 88.

El codi mostrat a la figura 88 el que fa és iterar sobre la llista de noms d'usuari i buscar-los a Twitter. Com es pot veure a la línia 8, imposem vàries restriccions sobre els usuaris. No volem usuaris que portin menys de 3 mesos registrats, o amb més d'un milió de seguidors, o que no hagin fet cap Tweet. Cada instància d'usuari que obtenim de Twitter la guardem a un *ArrayList* anomenat *users*, que és una variable global.

Amb els objectes d'usuari que hem obtingut de Twitter podem agafar la informació bàsica com per exemple el nombre de seguidors de l'usuari, però també volem agafar altres mètriques com la mitja de Tweets per segon. Per

[3] A l'hora d'obtenir els usuaris, aquesta taula contenia aproximadament 3 milions de Tweets

```
ResponseList l = null;
l = twitter.lookupUsers(userNames);
Calendar cal = Calendar.getInstance();
cal.add(Calendar.MONTH, -3);
Date threeMonthsBefore = cal.getTime();
for (i = 0; i < l.size(); i++) {
    User user = (User) l.get(i);
    if (user.getCreatedAt().before(threeMonthsBefore)
            && user.getStatusesCount() > 1
            && user.getFollowersCount() < 1000000) {
        users.add(user);
    }
}
```

Listing 88: Codi per a obtenir els usuaris de Twitter

a obtenir aquesta mètrica ens cal fer una nova cerca dels Tweets realitzats en els últims 3 mesos. Per tant ara el que cal fer és iterar sobre la llista *users*, obtenir aquestes mètriques i desar-les a Cassandra. Per facilitat a l'hora de llegir el document he obviat el codi, però és pot trobar tot al cd adjunt.

Limitacions

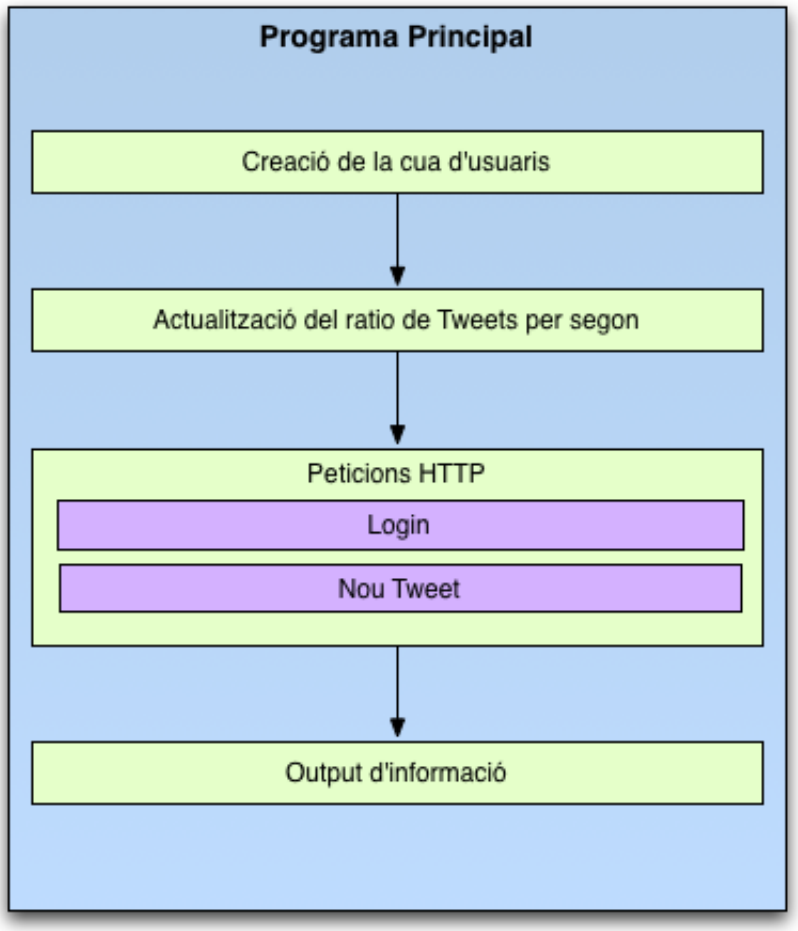
Com hem vist quan parlàvem sobre l'API de Twitter, existeixen limitacions horàries sobre el nombre de consultes permeses. Per tant tot el codi explicat a aquest capítol va englobat en funcions que tenen en compte aquestes limitacions, i un cop assolida la limitació es "re-programen" per tornar a començar la cerca quan es torni a tenir permís per a fer noves crides. Això fa que el codi es pugui deixar funcionant sense haver-nos de preocupar de res.

Per altra banda, la crida per a obtenir els usuaris és una crida protegida, i per tant abans de fer cap crida ens cal autentificar les peticions amb el codi mostrat anteriorment a la figura 73.

6.2 Sistema de *benchmarking* per a Twissandra

Ara ja hem vist de quines mètriques disposem i com ho hem fet per a obtenir-les. Passem a veure el sistema de *benchmarking* implementat per a fer les proves. j

6.2.1 Estructura



Listing 89: Estructura del programa de *benchmarking*

A la figura 89 es mostra l'estructura del sistema de *benchamrking* implementat. Aquest porgrama consta d'un mètode principal que s'encarrega de crear 4 threads diferents (mostrats en verd a la figura).

El primer thread és l'encarregat de crear una llista ordenada amb els usuaris que han de fer els Tweets. Aquesta llista es crea usant el ratio de temps entre Tweet de cada usuari. Cada vegada que aquesta llista conté menys de 100 usuaris, es torna a activar el thread i s'afegeixen els pròxims usuaris que han d'enviar un Tweet a la llista.

El segon thread s'encarrega d'actualitzar el nombre de Tweets per segon

que corresponen a l'instant de temps actual segons la funció obtinguda i explicada anteriorment. Aquest thread s'executa cada 2 minuts per defecte però es pot modificar aquest valor si es creu necessari.

El tercer Thread, és l'encarregat de realitzar les peticions a Twissandra. Es poden fer dos tipus de peticions, el login i l'enviament del Tweet. La funció de login retorna una *cookie* amb la informació de la sessió corresponent a l'usuari que fa el login, i per tant, per raons de rendiment el que es fa és desar aquesta informació. Així només cal que fem el login de cada usuari una vegada al llarg del *benchmark*. La segona petició que es fa és la d'enviar un Tweet, simplement s'envia un text aleatori amb la llargada que ens indica el valor de la llargada de Tweet mitja corresponent a cada usuari.

Aquest thread és el que crea el coll d'ampolla al client, si no féssim aquestes peticions HTTP en un thread, el programa es bloquejaria fins a que s'acabessin les peticions, i per tant no podriem garantir que es fan el nombre de peticions corresponent a cada segon. Així que es creen tants threads de peticions HTTP com peticions per segon corresponen a un moment de temps determinat. El problema que té aquest funcionament, es que si el servidor no es capaç de resoldre totes les peticions que li enviem, el client de *benchmarking* anirà acumulant threads i arribarà un punt en que no en podrà generar més i acabarà l'execució.

Per últim existeix un altre thread que cada cert nombre de segons imprimeix per pantalla informació del sistema. Així com el nombre d'usuaris que conté la cua explicada al primer thread, o el nombre de peticions realitzades i el nombre de peticions acabades, que ens indica el nombre de peticions que té pendents d'acabar el servidor.

6.2.2 Implementació

Acabem de veure l'estructura del client de *benchmarking*, aquest client, està implementat en la seva totalitat en java. Passem a veure'n les parts més rellevants.

Llista d'usuaris

El primer element del client, és una llista ordenada que conté una llista ordenada amb els usuaris que han de fer els pròxims Tweets. Per a construir aquesta llista s'ha fet servir una *ConcurrentLinkedQueue* de Java. Aquest tipus d'estructura de dades ens ofereix una llista ordenada del tipus "first in first out" amb la particularitat de que es "thread safe", la podem fer servir a dos threads diferents, i no existeix la possibilitat d'error. Això és molt

important ja que les escriptures es fan a un thread però les lectures a un altre.

El mètode per crear la llista és molt senzill, consta de dues petites funcions.

```
public void nextSecond(){
    synchronized (this) {
        seconds++;
        Long mul = seconds * ratio;
        for(User user : users){
            if(user.hasToPost(mul)){
                user.hasPosted();
                queue.add(user);
            }
        }
    }
}
```

Listing 90: Funció que construeix la llista amb els pròxims usuaris a enviar un Tweet

El que fa la funció és simular que passa un segon, aleshores mira la llista d'usuaris del sistema i determina per a cada un si ha de fer un Tweet en aquell segon o no. Això ho fa mitjançant una funció de la classe usuari que veurem a continuació.

```
public boolean hasToPost(long time){
    return time / ratio > posted;
}
```

Listing 91: Funció que determina si un usuari ha d'enviar un Tweet

La funció 91 pertany a la classe usuari, i determina si aquest ha d'escriure un Tweet o no a un segon determinat. Per a fer-ho s'agafa els segons transcorreguts (simuladament) i es divideixen pel nombre de segons entre Tweets d'aquell usuari, si el resultat és major al nombre de Tweets enviats per aquell usuari, això vol dir que li correspon enviar un Tweet a aquell segon.

Com que la informació que disposem de les mètriques agafades de Twitter és el nombre de segons entre Tweet d'usuari, i això implicaria haver de simular molt segons ja que el més normal és que el que enviï més Tweets ho faci cada varis minuts, el que es fa a la funció de la figura 90 és multiplicar un segon

per una variable *ratio*, que conté el mínim nombre de segons entre Tweets dels usuaris del sistema. Així ens estalviem vàries simulacions que no ens donarien cap resultat. Amb aquesta modificació ens assegurem que cada simulació de segon com a mínim ens donarà un usuari.

Ratio de Tweets per segon

Per a calcular el ratio de Tweets per segon corresponents a un instant de temps, l'únic que hem de fer és obtenir el ratio corresponent a l'instant de temps segons la funció obtinguda del model real de Twitter.

```
1  private static double ratioForTime(Date time) {
2      Calendar cal = Calendar.getInstance();
3      cal.setTime(time);
4      int hour = cal.get(Calendar.HOUR);
5      int minute = cal.get(Calendar.MINUTE);
6      int sum = hour * 3600 + minute * 60;
7      double x = sum / 86400;
8      double[] a = {8.4264, 12.392, 105.35, -1418.4,
9          4128.3, -4564.4, 1738, 5};
10     double y = a[0] + x * (a[1] + x * (a[2] +
11         x * (a[3] + x * (a[4] + x * (a[5] + x * (a[6]))))));
12     return y;
13 }
```

Listing 92: Funció que determina el nombre de Tweets corresponents a un instant de temps

La funció de la figura 92 calcula el nombre de Tweets corresponents a un instant de temps segons la funció obtinguda amb el model de Twitter. Com que aquesta funció s'ha obtingut amb el Microsoft Excel, cal fer una transformació abans de cridar a la funció. Aquesta transformació es deguda a que Excel representa els instants de temps en una escala de 0 a 1, on 0 equival a les 00:00:00 i 1 a les 23:59:59. Per tant es fa aquesta transformació i després es passa per la funció polinòmica que calcula el valor corresponent.

Peticions HTTP

Aquesta part és la més conflictiva ja que és la crea els colls d'ampolla, o bé al servidor que executa el *benchmark* o bé al servidor que executa Twissandra. Primer de tot vaig implementar les funcions usant una llibreria d'Apache, però era una mica massa complexe i por eficient, com a mínim tal i com

la vaig implementar jo. Aleshores vaig optar per fer crides des de Java a la comanda *curl* de *Unix*, però tampoc era massa eficient ja que el fet d'executar comandes del sistema operatiu des de Java no és massa òptim. Per últim vaig optar per fer servir el codi més senzill, la classe *URLConnection* de Java. Com que jo necessitava agafar les *cookies* amb la sessió de l'usuari el codi va quedar una mica enrevessat i no el poso al document. Tot i que el funcionament és absolutament trivial, fa una consulta POST a una URL i li passa unes dades. El nom de l'usuari i el password en el cas del login, i el cos del Tweet en el cas d'enviar un Tweet.

A part de l'implementació de les crides, el segon problema que va sorgir era que s'havia de mantenir el nombre de peticions per segon indicat pel ratio obtingut a la funció de l'apartat anterior. Com que les crides HTTP que ofereix la classe usada per a fer les peticions, son síncrones, aquestes es bloquegen fins a que obtenen una resposta del servidor. Això fa que no puguem garantir el nombre de peticions per segon, ja que si les crides triguen molt (la majoria triguen mig segon o més), el client es quedarà bloquejat fins a que aquestes acabin i no podrà totes les que corresponen a cada segon. Per a solucionar aquest problema hi havia dues solucions, usar crides asíncrones, o fer les crides en Threads individuals. Com que amb la classe *URLConnection* no semblava factible fer les crides asíncrones, em vaig decantar per fer un thread per a cada consulta.

El fet de crear un thread per a cada consulta implica que si el servidor web no les resol totes, es van acumulant threads al client que fa el benchamrk. I això no té cap solució, ja que limitar el nombre de crides implica no produir la càrrega corresponent a cada segon.

El codi que fa les crides HTTP és el següent:

```
private static void postTweets() {
    new java.util.Timer().schedule(
        new java.util.TimerTask() {
            @Override
            public void run() {
                for (int i = 0; i < ratio; i++) {
                    final User user = usersQueue.poll();
                    try {
                        if (!cookies.containsKey(user.getId())) {
                            login(user);
                        }
                        (new Thread(new Runnable() {
                            public void run() {
                                postTweet(user);
                            }
                        })).start();
                    } catch (Exception ex) {}
                }
                postTweets();
            }
        },
    1000);
}
```

Listing 93: Funció per a generar les peticions HTTP

Com es pot veure, aquesta és una funció que s'executa dins d'un timer que es programa per a executar-se cada segon.

6.2.3 Realització de les proves

Preparació de l'entorn

Abans de poder executar les proves, cal preparar l'entorn. Bàsicament el que ens cal és carregar totes les dades necessàries a Twissandra. Per a fer-ho disposem de dos *scripts* en Python.

El primer d'ells es diu createUsers.py, i el que fa és obtenir tots els usuaris que hem obtingut de Twitter i crear-los a Twissandra. Com hem vist, el més important és que cada usuari tingui definit el seu nombre de seguidors. Això ens porta al segon *script*.

El segon *script* s'anomena addFriends.py, i el que fa és crear un milió d'usuaris destinats a ser seguidors dels usuaris que faran els Tweets.

A la figura 94 veiem l'*script* per a crear els usuaris que faran de seguidor. Simplement és un bucle que crida a la funció *save_user* de Twissandra.

```
j = 0
while(j < 1000000):
    j = j + 1
    cass.save_user('follower'+str(j), {
        'password': 'pass',
    })
```

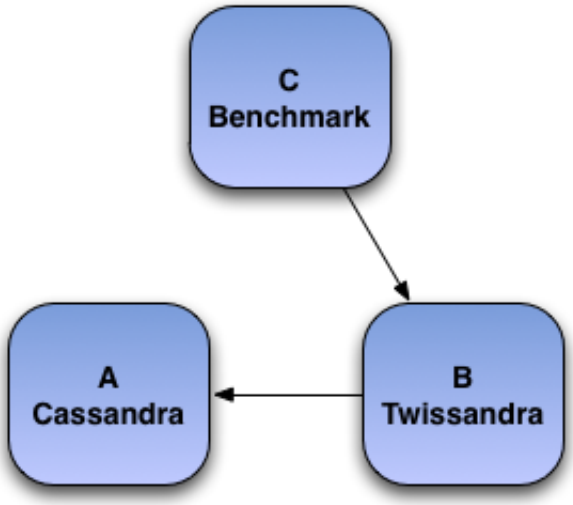
Listing 94: createUsers.py

Per al correcte funcionament dels tests, també cal comentar unes línies als fitxers de Twissandra, que fan re-direccions després de fer el login i després d'escriure un Tweet. Aquestes re-direccions fan que no puguem obtenir les sessions dels usuaris i per tant no puguem fer cap Tweet. En concret s'han de comentar les línies 27-29 del fitxer users/views.py i la línia 22 del fitxer tweets/views.py.

Entorn de test

Per a fer els tests es disposava de 3 servidors amb les característiques que es poden veure a la figura 95. En un inici es va decidir separar els servidors per capes i posar una capa del *benchmarking* a la màquina menys potent, el servidor de Twissandra la segona menys potent i Cassandra al més potent de tots. La distribució es va fer així ja que la intenció era produir el màxim de càrrega possible sobre Cassandra, i previsiblement el servidor de Twissandra hauria de processar una gran quantitat de peticions.

Inicialment s'havia de disposar d'un segon servidor "clonat" d'A, exactament igual, al que es volia posar un segon node de Cassandra per a fer proves amb un clúster real de Cassandra i poder provar diferents models de consistència. Per raons tècniques no es va poder disposar d'aquest servidor.

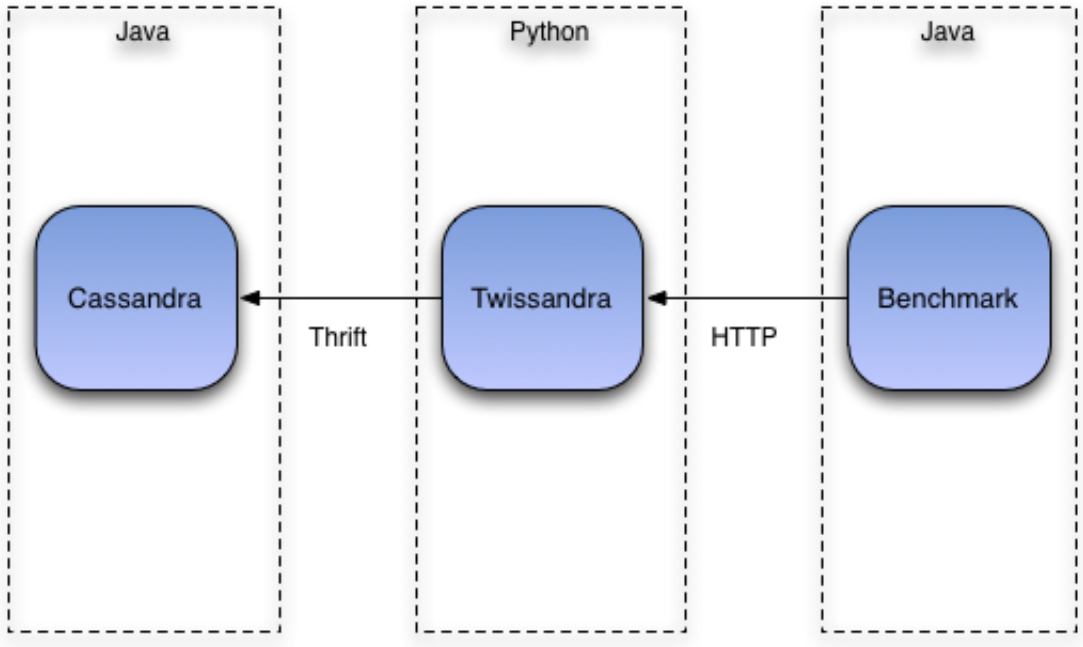


Màquina	Processadors	Familia	Freqüència	Memòria
A	8	Xeon	2.83 GHz	16 GB
B	8	Xeon	2.60 GHz	16 GB
C	4	Xeon	3.16 GHz	16 GB

Listing 95: Esquema de l'entorn de Test

Execució

Amb tot preparat ja es podien fer les primeres execucions per a obtenir resultats. A l'hora de fer els tests, però, van sorgir alguns problemes. Fixem-nos en detall amb les capes que recorre una petició d'inserció:



Listing 96: Capes que travessa una petició

Com podem observar, hi ha moltes capes, i cadascuna d'elles requereix un entorn diferent, la primera part és Java, després tenim una capa en Python, i per últim tornem a Java passant per Thrift. Per tant, el que va passar al fer les proves va ser que ens vam topar amb un coll d'ampolla introduït per software.

En concret, el servidor web a on es feia servir Twissandra. Django [4] ofereix un petit servidor de proves per a testejar les aplicacions. I les proves es van fer sobre aquest servidor. El problema va ser que que aquest servidor ràpidament es saturava i impedia realitzar els tests.

Aleshores es va optar per traslladar l'entorn de test als servidors del departament del director del projecte. Això va millorar significativament els temps de resposta, però encara no era suficient per a poder realitzar els tests desitjats. Els millors resultats obtinguts realitzaven vàries peticions sense problema, però el més normal era que quan ja s'havien llançat 60 peticions, encara en faltessin per processar 10. Per tant, amb tant poques peticions realitzades, arrossegar un nombre de peticions tant elevat, deixava veure que seria inviable arribar al mínim nivell desitjat que eren les 5 peticions per

[4] El framework amb el que està desenvolupat Twissandra

segon mínimes del Twitter real.

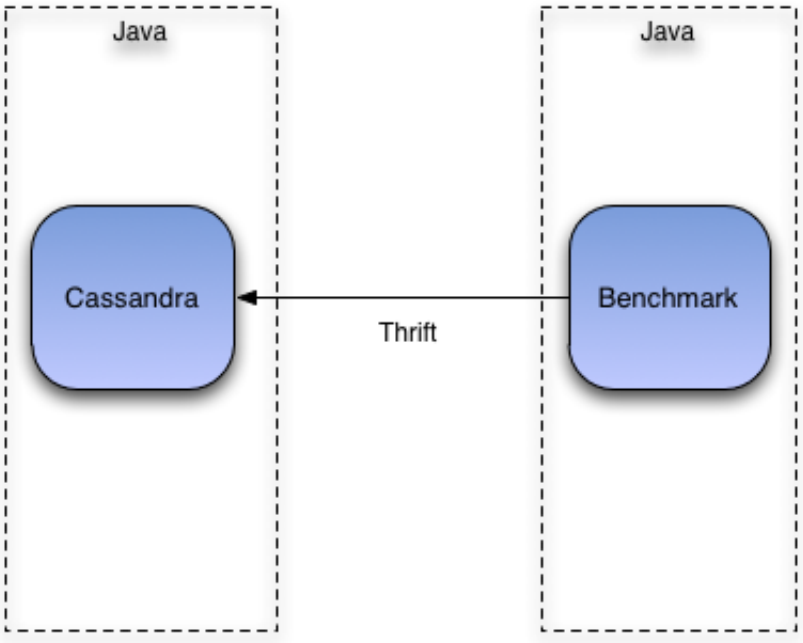
Com que aquestes limitacions venien imposades per capes software, el que calia era refer el sistema de testing, canviar el servidor web i segurament retocar alguna de les altres parts. Com que el temps disponible era molt limitat es va optar per reconduir l'estratègia i fer els tests d'una altra manera.

6.3 Proves de càrrega sobre Cassandra

Com que les proves eren l'últim punt del projecte, tots aquests entrebancs van fer que no quedés marge de rectificació, no hi havia temps d'adaptar totes les parts per a poder fer les proves. Aleshores vaig decidir aprofitar tota la feina feta, per com a mínim realitzar unes petites proves i mirar d'avaluar el rendiment de Cassandra amb diferents configuracions.

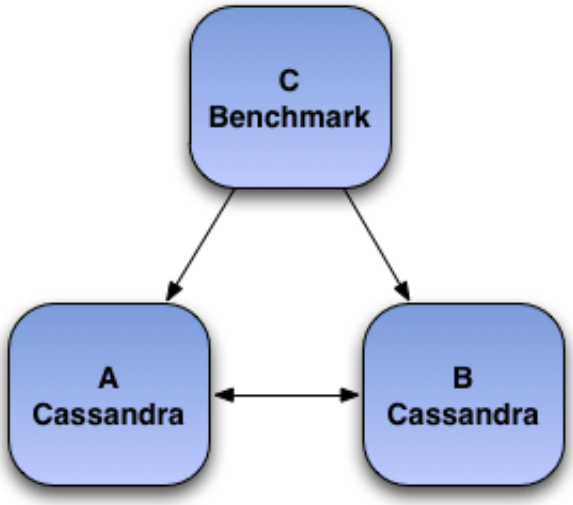
6.3.1 Entorn de proves

El primer pas va ser eliminar totes les capes intermitges. Passar directament del client de *Benchmarking* al servidor de bases de dades:



Listing 97: Nou entorn d'execució

Una vegada fet aquest re-disseny, es van re-assignar les parts als servidors disponibles:

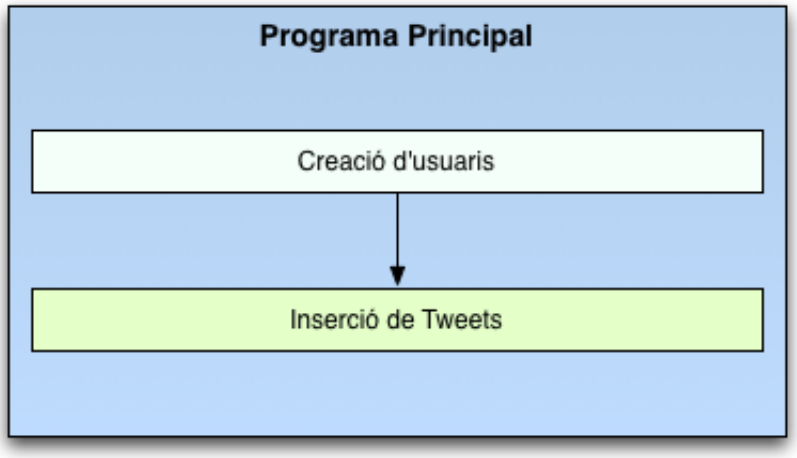


Listing 98: Esquema de l'entorn de Test

Es va assignar els dos servidors més potents i amb les característiques més similars per a formar el clúster de Cassandra, mentre que el tercer servidor seguiria sent l'encarregat de realitzar les proves.

6.3.2 Test de càrrega

El test que es va plantejar era un test molt més simple. El fons del test era el mateix, inserir Tweets per a veure el comportament de Cassandra. Així que es va agafar el codi ja existent i es va simplificar:



Listing 99: Esquema de l'entorn de Test

Com es veu a la figura 99, ara el programa només tenia dues parts. Primer creava un conjunt d'usuaris a la base de dades, com a preparació per als tests. Tot seguit es passa directament a fer les insercions. Per a optimitzar el rendiment, es crea un thread per a cada usuari, i el thread únicament fa les insercions a Cassandra.

El que si que es va haver de fer va ser optimitzar una mica el codi. El que volia era provar el rendiment de Cassandra, per això havia tret totes les capes intermitges. Amb les primeres execucions vaig veure que es feien masses crides entre el client i el servidor. Ja que per a cada Tweet havíem de fer $3 + n$ insercions, com hem vist a capítols anteriors. Això significa que es feien per cada Tweet $3 + n$ crides al servidor. El temps que es perdia en comunicació entre servidors feia variara molt els resultats. Al final vaig descobrir que Cassandra dona una opció de realitzar múltiples insercions en una sola crida, amb el següent codi:

```
1  private static void postTweet(User user) {
2      Mutator<String> mutator = HFactory.createMutator(keyspace, se);
3      UUID uid = UUID.randomUUID();
4      Long ts = System.nanoTime() / 1000;
5      LongSerializer ls = new LongSerializer();
6      mutator.addInsertion(uid.toString(), "Tweet",
7          HFactory.createStringColumn("body", body.substring(0,
8              user.getAvgLength()))).addInsertion(uid.toString(), "Tweet",
9          HFactory.createStringColumn("username", user.getId()));
10     mutator.addInsertion(user.getId(), "Userline",
11         HFactory.createColumn((Long) ts, uid.toString(), ls, se));
12     mutator.addInsertion("!PUBLIC!", "Userline",
13         HFactory.createColumn((Long) ts, uid.toString(), ls, se));
14     int firstFollower = randomGenerator.nextInt(maxRandom);
15     for (int i = firstFollower; i < firstFollower + numFollowers; i++) {
16         mutator.addInsertion("Follower" + i, "Timeline",
17             HFactory.createColumn((Long) ts, uid.toString(), ls, se));
18     }
19     try {
20         mutator.execute();
21     } catch (Exception ex) {}
22 }
```

Listing 100: Enviament d'un Tweet amb una única crida

Si ens fixem en el codi de la figura 100, es pot veure com es van afegit operacions d'escriptura a un objecte *Mutator*, i al final es fa una única crida a *mutator.execute()*. Amb aquest codi ara ja només es fa una crida al servidor per cada Tweet, en comptes de $3 + n$. Per tant ara ja si que podem avaluar de forma més fidedigne el rendiment de Cassandra.

Aquesta nova versió de programa de *Benchmarking* requereix 3 paràmetres:

- Nombre d'usuaris
- Nombre de seguidors
- Nombre de Tweets

Amb aquests paràmetres es defineix el nombre d'usuaris que faran les insercions, quants Tweets farà cadascú, i quants seguidors tindran els usuaris. Tots els usuaris seran exactament iguals. D'aquesta manera podem controlar que sempre executarem el mateix nombre d'insercions i per tant podrem comparar els resultats obtinguts. Si agaféssim usuaris aleatoris amb diferents nombres de seguidors, no podríem comparar els resultats, ja que no podríem assegurar que l'execució ha estat igual.

6.3.3 Execucions i resultats

Una vegada el codi va estar suficientment optimitzat i garantia que les proves avaluarien el màxim possible el rendiment de Cassandra i no dependrien de cap capa intermitja es van realitzar els tests.

Després de fer unes quantes proves, es va fixar els paràmetres d'entrada al programa en:

- Nombre d'usuaris: 10
- Nombre de seguidors: 5000
- Nombre de Tweets: 100

Aquests paràmetres ofereixen un balanç entre usuaris (threads) i nombre d'insercions (5000 per tweet), que permeten executar proves significatives amb temps d'execució raonables. Cal fixar-se que tot i que no es fan $3 + n$ crides per Tweet, si que es fan $3 + n$ insercions per Tweet. El nombre total d'insercions que es fan sobre Cassandra amb aquests paràmetres son 2.003.000.

Amb aquests paràmetres fixats, es van realitzar 3 execucions per cadascun dels entorns que veurem a continuació. Per a totes les proves es fa servir el particionador aleatori.

6.3.4 Un node

Primer vam posar un únic node de Cassandra i vam realitzar 3 execucions. El temps d'execució mitjà va ser de $72.218,67$ milisegons, i per tant el cost mitjà per inserció és $0,036$ segons.

6.3.5 Dos nodes

Tot seguit es va afegir un segon node. No es va tocar cap paràmetre de la configuració, per tant el factor de rèplica era $N = 1$. Després de les 3 execucions es va obtenir un temps d'execució mitjà de 67507 milisegons, amb un temps mig per inserció de $0,0337$ milisegons.

Sorprenentment, el temps mig per inserció és inferior a quan teníem un node. Jo esperava obtenir temps clarament superiors. Si pensem que tenim el particionador aleatori i per tant el nombre d'insercions per node ha de ser gairebé igual seria lògic veure una baixada de rendiment degut al transport de peticions entre els dos servidors de Cassandra. Vaig fer les comprovacions pertinents, i realment les insercions estaven equilibrades entre els dos nodes.

6.3.6 Dos nodes amb N=2

El següent pas va ser incrementar el factor de rèplica a 2 ($N = 2$), és a dir, ara cada inserció s'havia de fer als dos nodes, ja que al haver-hi únicament dos nodes i factor de rèplica de 2, cada node ha de tenir una còpia de totes les dades. Els resultats van donar un temps mitjà d'execució de $171.425,67$ milisegons, amb un temps mig per inserció de $0,0856$ milisegons.

Per tant, l'increment del factor de rèplica si que té implicació directe al cost de cada inserció, una inserció passa a trigar més del doble que quan teníem el nombre de rèpliques a 1.

6.3.7 Valor de consistència

Després de veure que al augmentar el factor de replicació augmentava el cost de cada inserció, podíem deduir que cada petició d'escriptura es quedava bloquejada fins a que es fessin les dues insercions. Per tant ara calia provar de modificar el nivell de consistència de les escriptures.

S'havien de fer proves amb $W = 1$, $W = 2$ i $W = 0$. És a dir, esperar només a que es fes l'escriptura a un node, esperar que es fes als dos, o no esperar a cap confirmació, el que es diu escriptures asíncrones.

Es van realitzar diferents proves, però els resultats sempre eren iguals als obtinguts a l'apartat anterior, per tant he de creure que el codi que vaig fer servir per a modificar aquests paràmetres no tenia cap influència real a l'hora de les peticions.

6.3.8 Observacions realitzades

Al fer diferents execucions, van succeir diferents esdeveniments que crec que son interessants d'explicar.

Una de les execucions va trigar 3 vegades més del que havien trigat la resta de les execucions amb els mateixos valors d'entrada, a priori no tenia cap sentit i vaig decidir mirar quina podia haver estat la causa d'aquest comportament. Al final vaig descobrir que durant aquella execució, un dels nodes de Cassandra havia arribat al límit de mida per a la majoria de les seves SSTable, per tant havia estat realitzant moltíssimes més escriptures a disc mentre s'inserien els Tweets al commitLog.

Una altra de les execucions quan es tenia factor de rèplica $N = 2$, va trigar la meitat que la resta. Va donar els mateixos resultats que quan es tenia factor de rèplica $N = 1$. Mirant els logs vaig veure que un dels nodes havia exhaurit l'espai al disc, i per tant totes les escriptures s'estaven fent a un únic node, per això els valors eren iguals a quan es tenia $N = 1$.

6.3.9 Conclusions

Amb aquests tests vaig poder veure com diferents configuracions tenien diferents rendiments, tot i que em va quedar pendent realitzar les proves amb diferents factors de consistència.

El que més em va sorprendre va ser el fet de que no incrementés el temps d'inserció al afegir un segon node. m'hagués agradat fer proves amb més servidors, però trobo sorprenent que es mantinguin els temps quan es treballa amb dos servidors. Segurament em sorpren perquè vaig provar de fer unes proves fa bastant temps a casa amb dos ordinadors i els temps van créixer molt al posar un segon node. Això em fa veure la rellevància que té treballar amb màquines potents i amb bona infraestructura de xarxa.

Per altra banda m'ha agradat veure que Cassandra tal i com promet sempre permet fer les escriptures. A l'execució en que un node es va quedar sense disc, no es va veure afectada per a res l'execució de les proves.

Per últim veig que les proves permeten veure com diferents configuracions afecten al rendiment de Cassandra, però no permeten extreure conclusions sobre el rendiment real del sistema. Ja que per exemple, l'execució que va coincidir amb totes les compactacions es va veure realment afectada. Tot i que cal dir que a la resta d'execucions, ha estat freqüent la realització d'alguna compactació entremig. Però per a avaluar realment el rendiment de Cassandra caldria fer proves més llargues, forçant compactacions, i sobretot amb més nodes i molta més càrrega.

6.4 Instrumentació Twissandra

La idea principal a l'hora de fer els tests era realitzar totes les proves sobre Twissandra. Però amb la peculiaritat de tenir Twissandra instrumentat per a desar logs sobre HBase per a després poder realitzar anàlisis dels logs. Això ens permetia provar Cassandra com a base de dades més dinàmica per a donar suport a una aplicació, i HBase com a sistema d'anàlisi de dades.

Així que el que es va fer va ser afegir unes petites funcions sobre Twissandra per a desar logs sobre HBase.

6.4.1 Implementació

Per a poder desar logs sobre HBase, el codi necessari era trivial. Només calia una funció que ens permetés desar a una taula d'HBase informació. Tota la informació es desava sobre una mateixa taula anomenada "logs".

```
def logHbase(muts):
    global sessionId
    raw_ts = int(time.time() * 1e6)
    key = str(raw_ts)+str(sessionId)
    muts.append(Mutation(column="info:idSessio", value=str(sessionId)))
    muts.append(Mutation(column="info:timestamp", value=str(raw_ts)))
    client.mutateRow("logs", key, muts)
```

Listing 101: Funció per a desar logs sobre HBase

La funció de la figura 101 mostra el codi necessari per a desar informació a la taula "logs" d'HBase. Aquesta funció rep com a paràmetre la informació que es vol desar. D'aquesta manera el codi és el màxim de re-usable possible.

Ara només calia desar els logs als punts desitjats de l'aplicació. Per a començar es van instrumentar dues operacions, però afegir instrumentació a una nova operació era absolutament trivial.

```
muts.append(Mutation(column="info:tipus", value="altaUsuari"))
logHbase(muts)
```

Listing 102: Instrumentació de la funció save_user

```
muts = []
muts.append(Mutation(column="info:tipus", value="nouTweet"))
muts.append(Mutation(column="info:idTweet", value=str(tweet_id)))
muts.append(Mutation(column="info:idClient", value=str(user_id)))
logHbase(muts)
```

Listing 103: Instrumentació de la funció save_tweet

A les figures i veiem el codi afegit per a instrumentar les funcions "save_user" i "save_tweet" respectivament.

6.4.2 Pig

Una vegada HBase conté els logs, el que volem és poder-los analitzar, i per a fer-ho la millor solució és usar Hadoop. Com que el llenguatge de Hadoop és bastant complicat, existeixen eines com Hive o Pig. Aquestes eines ofereixen llenguatges més similars al SQL.

Com hem vist a l'apartat anterior van sorgir contratemps amb l'entorn de test, i això va fer que aquesta part del projecte no es pogués realitzar. La instrumentació funciona correctament, però la falta de dades i la falta de temps van impedir acabar aquest objectiu. Tot i això es van poder fer algunes proves amb Pig i el funcionament era correcte.

Capítol 7

Conclusions

7.1 Conclusions tècniques

Al llarg d'aquest projecte he pogut aprofundir tant en HBase com en Cassandra. Tot i que les dues eines comparteixen una estructura, que és la orientació a columnes i el model marcat per BigTable, son dos productes molt diferents tant en quant a implementació com a funcionament.

HBase és una eina que es va decantar molt des d'un inici cap a l'anàlisi de dades. El seu focus sempre ha estat la velocitat en les lectures i la integració amb Hadoop. Això l'ha encasellat ràpidament i mai ha estat un inconvenient per a HBase. Totes les característiques que s'han anat afegint amb el temps han buscat potenciar aquest objectiu d'HBase, i des d'un inici s'ha mantingut fidel a un model.

Mentre que Cassandra primer s'enfocava inicialment a aplicacions més dinàmiques, però ràpidament va voler oferir també integració amb Hadoop. Aquest enfoc poc definit i el seu peculiar enfoc en quant a consistència de les dades, fa que Cassandra oferís al iniciar el projecte un aspecte menys sòlid que HBase.

HBase des del meu punt de vista és més que una base de dades. És adoptar un nou model en varis aspectes. Una empresa que opti per HBase, segurament serà una empresa amb grans quantitats de dades, i amb molta necessitat d'anàlisi sobre les seves dades. Que principalment el que busca és usar Hadoop, i únicament usa HBase com a una eina per a poder realitzar escriptures sobre el sistema de dades de Hadoop. Per tant, l'ús d'HBase actual al mercat, va molt lligat a l'ús de Hadoop.

Des d'el meu punt de vista, una empresa que busqui treballar amb una eina NoSQL, però no tingui a priori interès per Hadoop, no es decantarà per usar HBase. Això és degut a que HBase és un sistema més complexe, reque-

reix mantenir un clúster de Hadoop, governat per un sistema de Zookeeper, i a sobre posar-hi HBase. Això és molt pràctic si volem usar Hadoop, en cas contrari, només comporta mals de cap.

Per altra banda existeixen eines com Cassandra que únicament son bases de dades. Busquen oferir el màxim de capacitats en quant a desar i obtenir les dades, no es preocupen del que vulguem fer després amb les dades. Això no vol dir que si usem Cassandra no podrem usar Hadoop, ja que Cassandra ofereix una altíssima integració amb Hadoop. El que passa és que Cassandra no limita el seu ús en cap moment a Hadoop. Cassandra és únicament Cassandra, una base de dades orientada a columnes.

Per tant si el que busquem únicament és un sistema de bases de dades orientat a columnes, segurament Cassandra o sistemes similars és el que busquem. Mentre que si el que busquem son sistemes d'analitzar grans quantitats de dades, i tenir una manera còmode i eficient per emmagatzemar-les, el millor és usar HBase.

De qualsevol manera per això, cal dir que tots dos productes son sistemes molt "joves" i en plena evolució. Des de que vaig començar el projcete han sortit vàries versions noves i a dia d'avui, tan HBase com Cassandra ja ofereixen l'ús de claus secundàries, per exemple. Per tant ambdós sistemes, estan encara evolucionant i definint-se i no serà fins d'aquí un temps quan treguin les primeres versions no "betas" que es podrà fer un anàlisi exhaustiu i encasellar els productes en un mercat concret.

Per altra banda, m'ha quedat molt clar que son bases de dades per a un ús molt específic i a una escala més gran del que la gran majoria de projectes o empreses necessiten. Per a que els sistemes funcionin bé, cal tenir una bona infraestructura Hardware i fer-les servir per a entorns que realment generin grans quantitats de dades i milers de peticions, ja que sinó no val la pena l'esforç. Com a mínim amb aquestes versions actuals.

7.2 Valoració personal

Si recapitulo i penso quins eren els meus objectius personals al començar el projecte, m'adono que aquest projecte ha complert amb escreix les meves expectatives. Volia un projecte innovador, que em permetés conèixer coses desconegudes per mi fins aquell moment. En aquest aspecte, he sobrepassat totes les expectatives, en tots els apartats del projecte he treballat amb coses desconegudes. He conegut el model orientat a columnes, i dos nous productes, HBase i Cassandra. També he aprés coses noves com l'API de Twitter o la programació de Threads en Java. I per altra part, he fet petites coses amb un llenguatge en que havia treballat molt poc, Python. Per tant totes les

parts d'aquest projecte han implicat adquirir nous coneixements. També volia un projecte on tingués llibertat per marcar els meus propis objectius i poder treballar lliurement sense imposicions. En aquest aspecte també s'han superat les meves expectatives. El director del projecte no només no m'ha imposat res, sinó que a sobre m'ha ajudat i m'ha ofert tots els mitjans al seu abast per a que pogués realitzar el projecte.

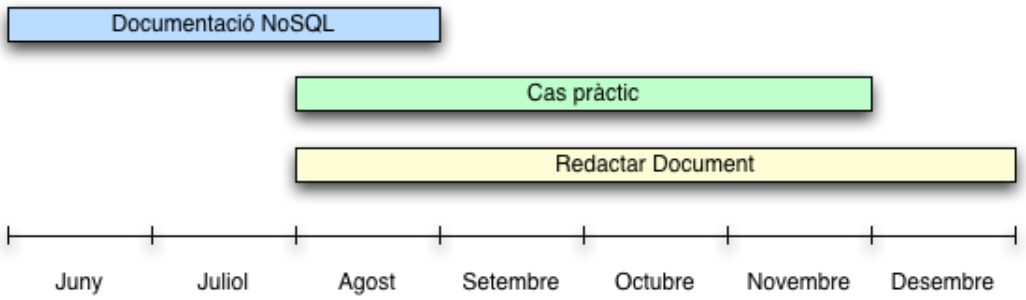
Per altra banda, el fet de desconèixer absolutament la majoria de continguts d'aquest projecte abans de començar, han tingut una implicació negativa en decisions equivocades i mal planning del temps. Això ha fet que no hagi pogut assolir tots els objectius que havia proposat i que m'hagi quedat amb ganes de poder realitzar tots els tests que m'hagués agradat fer. Però aquest aspecte també té una connotació positiva, i es que em quedo amb ganes de més, per tant vol dir que he gaudit fent el projecte i que moltíssimes hores després de començar, encaro segueixo tenint motivació per a seguir endavant amb aquest tema.

Realment l'únic punt negatiu que trobo a aquest projecte i el plantejament que n'he fet, ha estat que les parts que m'han portat més feina i que han fet que no hagi de tot el temps que m'hagués agradat tenir per a poder realitzar més proves, han estat les parts que no estan directament relacionades amb el projecte, com per exemple l'ús de l'API de Twitter, que ha estat de llarg la part que m'ha portat més mals de cap.

En definitiva, puc dir que he descobert un camp que realment m'interessa i al que vull seguir treballant. Faré tot el possible per a seguir treballant amb sistemes orientats a columnes als meus temps lliures, i a un futur pròxim quan hagi de buscar una feina, serà el camp a on intentaré situar-me. Per sort veig que és un camp que tot just està naixent i a on cada vegada més empreses es voldran dirigir, per tant espero poder-m'hi dedicar a la meva vida professional.

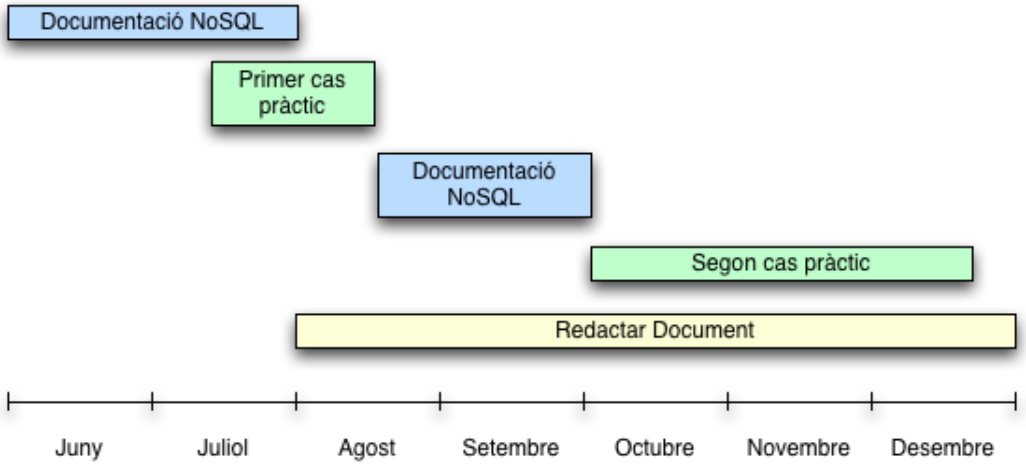
7.3 Diferències amb la planificació inicial

La planificació del projecte ha sofert moltes modificacions al llarg del projecte. Recordem la primera planificació:



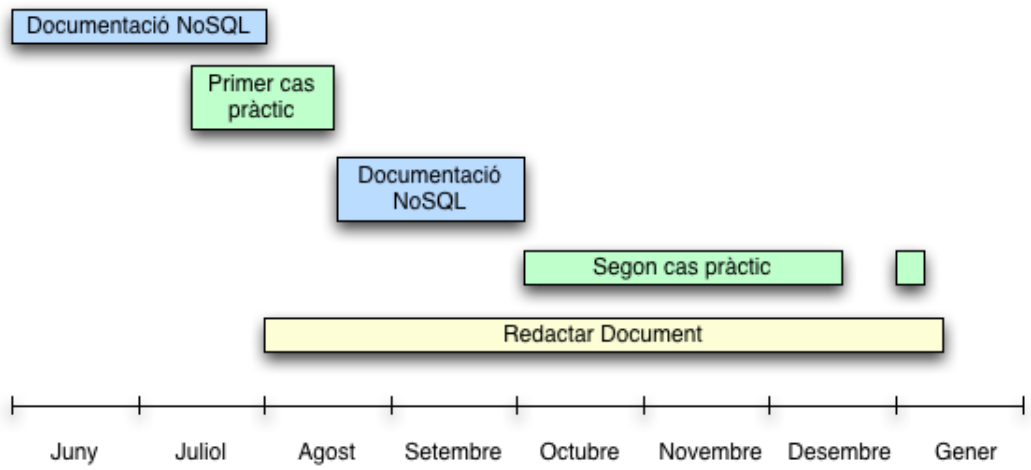
Listing 104: Planificació inicial del projecte

Com veiem, ni tan sols s'havia plantejat fer dos casos pràctics. La següent planificació va ser aquesta:



Listing 105: Planificació inicial del projecte

Aquesta planificació va ser una vegada vaig veure que el primer cas pràctic no estava prou ben enfocat i vaig decidir reprendre el procés de documentació i vaig plantejar un segon cas pràctic. Per últim fem un cop d'ull a com s'han separat les tasques realment durant la realització del projecte:



Listing 106: Planificació inicial del projecte

Com veiem és bastant fidel a la segona planificació, però s'ha allargat una mica i agafa part del Gener. A més el segon cas pràctic el vaig deixar de banda bona part del Desembre per a finalitzar la documentació, i una vegada acabada la majoria la documentació vaig reprendre el cas pràctic per a realitzar algunes proves i poder-les incloure a la documentació.

Aquests canvis a la planificació, reflecteixen clarament com aquest ha estat un projecte molt dinàmic, que s'ha anat re-definint constantment.

7.4 Objectius assolits

Si agafem de nou els objectius plantejats a la introducció, podem veure quins s'han assolit i quins no.

Comencem pels referents a la documentació, els objectius inicials eren:

1. Contextualització de les bases de dades orientades a columnes respecte els sistemes relacionals
   (a) Resum de les característiques dels sistemes relacionals
   (b) Explicació de les limitacions dels sistemes relacionals
   (c) Descripció dels sistemes orientats a columnes

Aquests objectius s'han assolit tots, en quan als sistemes relacionals per exemple, vaig acabar fent una documentació molt més detallada del que

inicialment havia plantejat. Això és degut a que vaig veure que calia posar més en context un sistema respecte de l'altre, ressaltant-ne els punts forts i els punts febles.

Els següents objectius eren:

2. Estudi dels sistemes de bases de dades orientades a columnes
   (a) Estudi de les característiques dels sistemes orientats a columnes
   (b) Estudi a fons d'HBase
   (c) Estudi a fons de Cassandra

Aquests objectius s'han assolit amb escreix. Per ser tecnologies que desconeixia totalment abans de començar, crec que vaig tenir molta sort amb la documentació que vaig acabar trobant i això em va permetre assolir un bon coneixement dels sistemes orientats a columnes, les seves arquitectures, els seus punts forts i els seus punts febles, i sobretot reflectir els coneixements a aquest document.

Per últim tenim els objectius tècnics:

3. Desenvolupament d'una aplicació per veure el funcionament de les bases de dades orientades a columnes.
   (a) Estudi del funcionament de Twissanra, una aplicació que funciona sobre Cassandra
   (b) Adaptació de Twissandra per a usar HBase
   (c) Instrumentació de Twissandra per a desar logs a HBase
   (d) Ús d'eines d'anàlisi de dades sobre HBase com Hive o Pig

Aquests objectius també els vaig realitzar tots excepte l'última part, la d'anàlisi de dades amb Pig. Tot i que vaig realitzar les diferents parts, una vegada finalitzades vaig veure que no era el millor enfoc que podia donar al projecte, i vaig decidir proposar nous objectius. Això fa que m'hagi quedat amb la sensació de que aquests objectius fossin menys importants, o que no m'hagin servit per al projecte, però la realitat es que em van ajudar molt a entendre les eines, i sobretot em van fer veure que m'havia de documentar més.

Tots aquests factors em van portar als últims objectius:

3. Desenvolupament d'una aplicació per veure el funcionament de les bases de dades orientades a columnes.

(a) Desenvolupar un generador de càrrega basat en Twitter sobre Twissandra

(b) Obtenir dades de Twitter mitjançant les API's

(c) Extreure models de comportament sobre les dades obtingudes de Twitter

(d) Generar càrrega seguint els models obtinguts

(e) Analitzar el rendiment de Cassandra, fent proves de càrrega sobre Twissandra

(f) Modificar el model de dades de Twissandra per a veure les diferències en el rendiment global de l'aplicació

(g) Analitzar els logs de Twissandra deixats sobre HBase

D'aquests objectius, no s'han realitzat els dos últims, i el de realitzar l'anàlisis de rendiment de Cassandra em quedo amb la sensació de no haver-lo assolit. Tot i que vaig realitzar tot el codi necessari i vaig començar a fer les proves, em quedo amb el mal gust de boca d'haver hagut d'abandonar les proves per problemes amb el Software i falta d'infraestructura.

M'hagués agradat disposar de més temps i poder finalitzar tots els objectius i realitzar totes les proves que havia plantejat inicialment. Però en general quedo molt satisfet amb el treball realitzat i espero poder-lo finalitzar algun dia o veure que algú reprèn la meva feina i la porta un pas més endavant.

7.5 Cost del Projecte

Si aquest projecte s'hagués de pressupostar a una empresa s'hauria de separar clarament en tres parts:

- Documentació
- Desenvolupament de Twitbase
- Desenvolupament de l'entorn de *Benchmarking*

La part de la documentació estaria orientada a elaborar un informe, com podrien ser els capítols 2, 3 i 4 d'aquest projecte. Això tindria un cost en hora aproximat de 300 hores. El nombre d'hores pot semblar molt elevat a priori, però s'ha de tenir en compte que es tracta de tecnologia desconeguda i que s'en elaborarà un document molt exhaustiu.

La segona i tercera part serien els desenvolupaments de les dues aplicacions que s'han presentat a aquest document com a casos pràctics.

Tasca	Hores	Responsable	Preu/hora	Total
Informe sobre sistemes orientats a columnes				
Documentació	200	Analista	30	6000
Redacció de l'informe	100	Analista	30	3000
Total	**300**			**9.000€**
Primer cas pràctic				
Planificació sistema	10	Cap de projecte	50	500
Estudi sobre Twissandra	40	Analista	30	1200
Programació HBase	100	Programador	20	2000
Total	**150**			**3.700€**
Segon cas pràctic				
Planificació sistema	24	Cap de projecte	50	1200
Especificació sistema	40	Analista	30	1200
Programació	150	Programador	20	3000
Realització tests	40	Programador	20	800
Total	**254**			**6.200€**
Cost Total Projecte	**704**			**18.900€**

Listing 107: Estudi de costos del projecte

A la figura 107, podem veure desglossats els diferents apartats que comporta cadascuna de les parts del projecte, el seu cost en hores i el responsable de realitzar cada tasca.

Si el client no disposés d'infraestructura i s'hagués de comprar o implementar sobre algun proveïdor de *Cloud Computing*, s'hauria de fer un pressupost separat, però de ben segur que tindria un coste elevat ja que com hem vist calen servidors potents per a obtenir una bona infraestructura i un bon rendiment.

7.6 Referències

Per a realitzar aquest document s'han fet servir moltes referències. Les presento a aquest apartat separades per temes.

7.6.1 MySQL

Llibre sobre alt rendiment amb MySQL [3]
Entrada a la Wikipedia sobre SQL [4]

7.6.2 NoSQL en general

Entrada a la Wikipedia sobre NoSQL [5]
Entrada a la Wikipedia sobre els sitmes orientats a columnes[6]
Entrada a la Wikipedia sobre BigTable [7]
Entrada a la Wikipedia sobre Denormalització [8]

7.6.3 HBase

Papers

Paper de presentació de BigTable [1]

Pàgines web

Entrada a la Wikipedia sobre HBase [9]
Entrada a la Wiki d'HBase sobre l'arquitectura interna d'HBase [10]
Article sobre test realitzats amb HBase per l'empresa Hstack [11]

Presentacions

Presentació sobre integració entre Hadoop i HBase [12]
Presentació amb varis casos pràctics. Parla sobre denormalització. [13]
Presentació sobre tests de rendiment amb HBase 0.20 [14]
Presentació sobre HBase i integració amb Java [15]
Presentació sobre integració d'HBase amb altres eines, com per exemple Pig [16]

Vídeos

Tot i que algunes de les presentacions incloses a l'apartat anterior també estan en format vídeo, vull fer una menció especial a una col·lecció de vídeos que han estat la font en la que realment m'he basat per a escriure la documentació. Sons uns vídeos dels que no puc posar l'enllaç, ja que son privats, pertanyen a un curs de formació d'una empresa anomenada Cloudera. Per un error tècnic van tenir els vídeos en obert uns dies i per casualitat hi vaig anar a parar. Quan vaig tornar per a revisar-los, havien desaparegut. Em vaig posar en contacte amb Cloudera, i els vaig explicar el meu projecte i els vaig demanar si m'hi podrien donar accés. Per sorpresa meva, van ser molt amables i em van donar accés als vídeos. Per tant, vull reiterar el meu agraïment a aquesta empresa ja que els seus vídeos son la base de la secció sobre HBase d'aquest document. [17]

7.6.4 Cassandra

Papers

Paper de presentació de Dynamo [2]
Paper sobre els protocols que fa servir Gossip [18]

Pàgines Web

Detallada explicació sobre Cassandra, i a la vegada comparació amb HBase [19]
Experiència de l'empresa CloudKick amb la migració a Cassandra [20]
Document format per 3 parts que explica molt detalladament Cassandra [21]
Document molt detallat del model de dades de Cassandra [22]
Wiki de Cassandra [23]

Presentacions

Cas pràctic de com una empresa (Digg) fa una migració dels seus sistemes cap a Cassandra [24]
Cas pràctic de com una empresa (Mahalo) fa servir Cassandra[24]

7.6.5 Material per als casos pràctics

Hector: Llibreria per a connectar amb Cassandra des de Java [25]
Twissandra [26]
Twitter4j: Llibreria per usar l'API de Twitter des de Java [27]

API de Twitter [28]
Guia per a aconseguir l'autenticació a Twitter amb Twitter4j i OAuth [29]

Bibliografia

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. C. nd Andrew Fikes, and R. E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*. Google, November 2006.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's Highly Available Key-value Store*. Amazon.com, 2007.

[3] B. Schwartz, P. Zaitsev, V. Tkachenko, J. D. Zawodny, A. Lentz, and D. J. Balling, *High Performance MySQL, Second Edition*. O'Reilly Media, June 2008.

[4] V. autors, "Sql." `http://en.wikipedia.org/wiki/SQL`.

[5] V. Autors, "Nosql." `http://en.wikipedia.org/wiki/NoSQL`.

[6] V. Autors, "Column-oriented dbms." `http://en.wikipedia.org/wiki/Column-oriented_DBMS`.

[7] V. Autors, "Bigtable." `http://en.wikipedia.org/wiki/BigTable`.

[8] V. Autors, "Denormalization." `http://en.wikipedia.org/wiki/Denormalization`.

[9] V. Autors, "Hbase." `http://en.wikipedia.org/wiki/HBase`.

[10] V. Autors, "Hbase architecture." `http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture`.

[11] A. Dragomir, "Hbase performance testing at hstack." `http://hstack.org/hbase-performance-testing/`, Abril 2010.

[12] T. Lipcon, "Apache hadoop and hbase." `http://www.slideshare.net/cloudera/tokyo-nosqlslidesonly`, Novembre 2010.

[13] J. Gray and M. Stack, "Practical hbase." http://wiki.apache.org/hadoop/HBase/HBasePresentations?action=AttachFile&do=view&target=ApacheCon2009_Practical_HBase-1.pdf, 2009.

[14] A. Rao and S. Zhang, "Hbase 0.20 performance evaluation." http://www.slideshare.net/schubertzhang/hbase-0200-performance-evaluation, Agost 2009.

[15] G. Helmling, "Hbase at stumbleupon." http://www.stumbleupon.com/devblog/hbase_at_stumbleupon/, Gener 2009.

[16] D. Ryaboy, "Pig, hbase, hadoop, and twitter: Hug talk slides." http://squarecog.wordpress.com/2010/05/20/pig-hbase-hadoop-and-twitter-hug-talk-slides/, Maig 2010.

[17] http://www.cloudera.com.

[18] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas, *Efficient Reconciliation and Flow Control for Anti-Entropy Protocols*. Amazon.com.

[19] D. Williams, "Hbase vs cassandra: why we moved." http://ria101.wordpress.com/2010/02/24/hbase-vs-cassandra-why-we-moved/, Febrer 2010.

[20] CloudKick, "4 months with cassandra, a love story." https://www.cloudkick.com/blog/2010/mar/02/4_months_with_cassandra/, Març 2010.

[21] O. Mallassi, "Let's play with cassandra...." http://blog.octo.com/en/nosql-lets-play-with-cassandra-part-13/, Juny 2010.

[22] A. Sarkissian, "Wtf is a supercolumn? an intro to the cassandra data model." http://arin.me/blog/wtf-is-a-supercolumn-cassandra-data-model, Setembre 2009.

[23] V. autors, "Cassandra wiki." http://wiki.apache.org/cassandra/.

[24] A. Sarkissian, "Looking to the future with cassandra." http://about.digg.com/blog/looking-future-cassandra.

[25] R. Tavory, "Hector." https://github.com/rantav/hector.

[26] E. Florenzano, "Twissandra." https://github.com/ericflo/twissandra.

[27] Y. Yamamoto, "Twitter4j." http://twitter4j.org/en/index.jsp.

[28] Twitter, "Api de twitter." http://dev.twitter.com/.

[29] M. Joseph and P. McCarthy, "Tweet your project's build status." http://www.ibm.com/developerworks/java/library/j-tweettask/index.html, Octubre 2010.